AUGUST 1989: VOLUME 2, ISSUE 5

Canada $4.25
USA $3.50

FOR THE Amiga Transactor

Amiga Network File System *by Doug Walker*

Adding 32-bit RAM To The A1000 *by Brad Fowles*

Animation Dynamics *by Cathryn and Eric Graham*

PLUS:

- Writing a compiler - part 2 in a series *by Chris Gray*
- VectorFonts in Modula-2 *by Anthony Bryant*
- Binary trees: three types *by Eric Giguére*
- Using the trackdisk.device *by Bob Rakosky*
- Regular columns by Jim Butterfield, Steve Ahlstrom and Don Curtis

Reviews: *FutureSound 500* digitizer, *Minimegs* RAM expansion, *ReSource* disassembler

Rayz by Derek Grime

THE MAGAZINE FOR AMIGA PROGRAMMERS

Lattice Tools & Libraries

We are the company that writes the language and the tools. So who could know more about the utilities you need to maximize productivity?

Lattice 5.0 Cross Compiler
All the power and facilities of the Lattice AmigaDOS 5.0 compiler in a cross-compiler MS-DOS and OS/2 host. Includes our full screen cross debugger. $750.

Lattice C++
Object Oriented C++ Programming for the Amiga. Provides object definitions for AmigaDOS, EXEC and Intuition. $300.

Lattice Compiler Companion
Collection of UNIX-like utilities that make your programming environment more productive. (Included in Lattice 5.0.) $100.

dBC III Library
Amiga programming library that provides programming interface into dBASE III compatible files. $150.

Communication Library
New Amiga programming library that supports modem interfaces for XMODEM, YMODEM, KERMIT and ASCII protocols. $250.

PANEL Library
Amiga programming library that supports windows graphics applications. $195.

C Programming Seminar
Learn C programming from Lattice experts. Four day seminar in Chicago. $895.

Call any of our world-wide distributors, or call us at 1-800-444-4309 or (312) 916-1600. Fax #(312) 916-1190, TELEX 532253.

"Brilliant!"

– William Hawes author of ARexx

"Lightning Fast!"

– Reichart Von Wolfsheild Silent Software, Inc.

"Blows away all the others."

– Robert Berryhill LMR Creations

Lattice AmigaDOS C 5.0 is getting lots of praise. And not just because of the latest benchmarks which prove Lattice C 5.0 is the best compiler by every measure.

For instance, Michael Berenstein calls 5.0 "Wonderful," while Roger Uzun simply calls this total programming environment "...the best."

Tom Dolan felt the same about CodePRobe, our new Source Level Debugger. Support for both C and Assembly language, plus multi-tasking debugging make it "...the best I have seen or used!!!!"

And our new Global Optimizer caused Steve Tibbett, a "VERY happy customer," to say, "WOW. This is outstanding!" and Tomas Rokicki of Radical Eye Software to exclaim "...Global Optimizer, inline subroutines and extensive libraries make whole new levels of performance possible."

About our manual? "Excellent!" And "...extra fine."

About our BBS for unlimited technical support? "Fantastic!"

And about our competition? Eric Dyke of Inom Software says, "Manx, eat your heart out! Lattice is the right choice!"

Lattice

Professional Programming Tools Since 1981
2500 S. Highland Ave., Lombard, IL 60148

InterComputing Inc.

2100 N. Hwy 360, Suite 2101
Grand Prairie, TX 75050-1015

In business since 1984; Amiga support since 1985
Member of Dallas Better Business Bureau

We are a service oriented business offering discount prices

FAX: 1-214-660-3695

Customer Service and Order Status call: 1-214-988-3500

COMPLETE PRODUCT LISTING AVAILABLE

Call toll free: **1-800-622-9177**

MEMORY EXPANSION

Amiga 500 & 1000

Starboard 0 K	A1000	239.95
Starboard 1 MB	A1000	499.95
Starboard 2 MB	A1000	795.00
Starboard 512 K	A1000	399.95
Starboard Multifunction Card		74.95
Starboard SCSI Module		99.95
Starboard Upper Deck		44.95
M501 0 K		59.95
M501 512 K with Clock		159.95
Mini Meg A1000 with 0.5 MB		359.95
Mini Meg A1000 with 1.0 MB		499.95
Mini Meg A1000 with 2.0 MB		789.95

Amiga 2000 & 2500

8-UP Memorycard	DIP	0K	159.95
8-UP Memorycard	DIP	2MB	595.00
8-UP Memorycard	DIP	4MB	995.00
8-UP Memorycard	DIP	6MB	1335.00
8-UP Memorycard	DIP	8MB	1795.00

VIDEO SOFTWARE

Artistic Expressions Vol. I & II each	39.95
Broadcast Titler	199.95
Comic Setter	66.95
Deluxe Paint 3.0	109.95
Deluxe Productions	144.95
Director	44.95
Director Toolkit	27.95
Fantavision	39.95
Innvision	169.95
Intro CAD 2.0	49.95
Movie Setter	69.95
Pageflipper Plus FX	99.95
Pagerender 3-D	109.95
Pagestream	139.95
Pagestream Fonts #1-5 each	29.95
Photon Paint 2.0	99.95
Professional Page	239.95
Sculpt 3D	69.95
Sculpt 4D	399.95
TV Show NTSC/PAL	62.95
TV-Text	62.95
Turbo Print	32.95
Turbo Silver	129.95
Zoetrope	99.95
Inovatools 1	

MUSIC & AUDIO

Audiomaster II	69.95
CMI Midi Interface	69.95
Dr. T's 4-OP Deluxe	99.95
Dr. T's Copyist	199.95
Dr, T's DX Heaven	99.95
Dr. T's KCS 1.6	166.95
Dr. T's Kawai K1 Lib.	99.95
Dr. T's Midi Recording Studio	44.95
ECE Midi-A500/2000	44.95
ECE Midi-A1000	44.95
"M"	154.95
Midi Cable 5' black	8.95
Midi Cable 8' color	11.95
Midi Gold external	69.95
Midi Gold internal	69.95
Music Mouse	49.95
Perfect Sound	64.95
Soundscape	
Audiodigitizer	79.95
Pro Midi Studio	124.95
Utilities I	39.95
Studio Magic	69.95
Synthia	66.95
Texture	

BUSINESS

Analyze 2.0	62.95
BEST Management System 2.0	299.95
Critic's Choice	159.95
Cygnus Ed Prof.	69.95
Excellence	179.95
Financial Plus	199.95
Kindwords	66.95
Maxi Plan 500	99.95
Maxi Plan Plus	129.95
Microfiche Filer	69.95
Microfiche Filer Plus	112.95
Money Mentor 2.0	59.95
Pen Pal	109.95
Phasar 3.0	66.95
Prowrite 2.0	72.95
Super Base Personal II	99.95
Super Base Prof. 3.0	199.95
Superplan	109.95
TexEd Plus	54.95
The Works-Platinum	169.95
Wordperfect	
Shakespeare	

HARD DRIVES

GVP Impact Drives:	
0MB-80meg Quantum	1199.95
0MB-No Drive	299.95
2MB-80meg Quantum	1699.95
2MB-No Drive	699.95
Hardframe DMA SCSI Controller	239.95
Phoenix Drives:	
22Meg A-1000	599.95
22Meg A-2000	549.95
32Meg A-1000	699.95
32Meg A-2000	749.95
48Meg A-1000	799.95
48Meg A-2000	749.95
80Meg A-1000	1299.95
80Meg A-2000	1249.95
Quantum Pro-80Meg	875.95
Supra 30MB Drive	699.95
Supra 30MB HDisk	699.95
Miniscribe	

FDATA-10 DRIVE

FDATA-20 DRIVE

Two 3.5" drives in one small case. Clean design, silent, and at a great price.

$ 2[illegible]

HEWLETT PACKARD INK-JET PRINTERS

HP-Deskjet 799.00

HP-Deskjet 999.00

DISK DRIVES

A2010 Internal 3.5"	149.95
AIR Drive External	169.95
AIR Drive Internal	134.95
California Access Drive CA-880	169.95
F-DATA 10	139.95
F-DATA 20	249.95
Drive Extension Cable	19.95

MODEMS

Supra 2400 baud ext.	149.95
Supra 2400 baud int.	149.95

VIDEO HARDWARE

A-Pro Draw 9x6	399.95
A-Pro Draw 12x12	499.95
Amiga Live A 1000	219.95
Amiga Live A 2000	329.95
Amiga Live A 500	299.95
Color Splitter	89.95
DigiDroid	64.95
Digi View 3.0	139.95
Digi View Gold	139.95
Easyl 500	319.95
Easyl 1000	339.95
Easyl 2000	349.95
Flicker Fixer	499.95
Panasonic 1410 Camera Kit	239.95
Perfect Vision	175.95
Polaroid Palette	2199.95
Super Gen	699.95
X-Specs 3D	99.95

PROGRAMMING

A/C Basic Compiler	129.95
A/C Fortran-AB Soft	179.95
Aztec C Developer	219.95
Aztec C Professional	159.95
Aztec Source Level Debugger	59.95
GFA Basic 3.0	99.95
Lattice C 5.0	219.95
Lattice C ++	399.95
Lint	69.95
Power Windows 2.5	56.95

MISCELLANEOUS

Arexx	32.95
BAD	32.95
CLI Mate	27.95
DOS-2-DOS	42.95
Disk Mechanic	56.95
EZ-Backup	32.95
Facc II	22.95
Project D	32.95
Quarterback	44.95

AMIGA 500 Starter Pack

A500 with 1Meg
2 Drives
Monochrome Monitor

949.95

Prepay with check or money order and get free UPS-ground shipping and a FREE GAME!

A-MAX 134.95

The 'MAC' Emulator for the Amiga works with all Amiga models

ROM-Set 139.95

For use with A-MAX

3.5" Drive 199.95

MAC-compatible; for use with A-MAX

TRADE UP!

We will trade in your Amiga 500 or 1000 towards a new Amiga 2000 or 2500. Call for details!

Government, corporate & institutional

Purchase Orders are welcome!

As always we have the most 'customer friendly' terms: S/H $3.95 in cont. USA; Minimum order is $30.00; MASTERCARD & VISA with NO credit card fee; in Texas add 7% Sales Tax. We ship to APO addresses, RMA# required on all returns. 18% restocking charge. All prices subject to change without notice.

Volume 2, Issue 5

Publisher
Antony Jacobson

Vice-President Operations
Jeannie Lawrence

Assistant Advertising Manager
Mike Grantham

Editors
Chris Zamara
Nick Sullivan
Tim Grantham

Assistant Editors
Malcolm O'Brien
Paul Bosacki

Contributing Writers
Steve Ahlstrom
Richie Bielak
Ben Blish
Daniel Brookshier
Anthony Bryant
Jim Butterfield
Betty Clay
Don Curtis
Matt Dillon
Andy Finkel
Brad Fowles
Eric Giguère
Cathryn Graham
Eric Graham
Chris Gray
Derek Grime
Jay Gross
Eric Haberfellner
Mike Hubbartt
James C. Lawrie
Rob Peck
Larry Phillips
Bob Rakosky
Dan Schein
Steve Simpson
Richard Stevens
John Toebes
Victor A. Wagner
Doug Walker
Clark Williams
James D. Williams
Doug Winger
David Wood

Cover Artist
Derek Grime

for the AMIGA Transactor

The Magazine for Amiga Programmers

AbsExecBase 4

Heartening news from this year's Devcon in San Francisco; and a new name on the *Transami* masthead.

A File System Case Study 26

by Doug Walker

Part 4 in our series on file system handlers: a network file system for the Amiga from the Software Distillery.

The FRANCES Memory Board 30

by Brad Fowles and Eric Haberfellner

Adding 32-bit memory to the LUCAS 68020/68881 accelerato rboard turns an Amiga 1000 into a 16-MHz killer workstation.

Dynamics For Animators, Part 2 38

by Cathryn and Eric Graham

Modelling the motion of elastic objects; and a program to read IFF scenefiles.

Toy: A Small Compiler For The Amiga 44

by Chris Gray

In Part 2, Chris reveals the operational details of the Toy compiler: its data structures, symbol table and tokenizer.

Copying Disks 54

by Bob Rakosky

Bypassing AmigaDOS and using the the trackdisk device directly provides faster, more flexible disk access. Bob's *Copydisk* program shows how.

A Walk In The Binary Forest 64

by Eric Giguère

The theory and practice of three tree types.

Programming In M2AMIGA 69

by Anthony Bryant

Vector-based fonts allow a programmer to easily scale and rotate text. Anthony shows how to use the Draw() function from M2AMIGA Modula-2.

Departments and Columns

Amiga Dispatches 6

by Don Curtis

Electrostatic discharge: the shocking truth! Plus: the latest flashes on the keyboard LED, an 11-Mb floppy and the A2090 hard drive controller.

ACCESS 10

by Steve Ahlstrom

Shareware, PD, freely distributable... what's the difference?
Plus: the latest utilites from Khalid Alsoseri.

Mastering The Raster 14

by Derek Grime

Some of DPaint III's most powerful capabilities are hiding under the slick surface.

Assembly Language Programming 18

by Jim Butterfield

Jim explains why the 68000 instruction set is not as orthogonal as some maintain - and that's about the size of it.

NewsLINK 80

Reviews

ReSource by Glen McDiarmid 75

FutureSound 500 from Applied Visions 78

MiniMegs RAM from M.A.S.T. 79

About the cover: *Rayz* by Derek Grime

Rayz was created with *Deluxe Paint III*, using extra-halfbrite mode to provide the varied colouring in the hair. The sun in the picture was created with concentric circles and the 'smear' function. The picture was converted to HAM mode with *DigiView Gold* and imported into *Professional Page v1.2*. Derek runs his own graphics company, Beyond Graphics, and is the author of the regular **Mastering the Raster** column in *Transami*. *Rayz* is his third cover.

Transactor for the Amiga is published monthly by Croftward Publishing Inc., 10-85 West Wilmot Street, Richmond Hill, Ontario, L4B 1K7. ISSN# 0838-0163. Canadian Second Class Mail Registration No. 7690, Gateway-Mississauga, Ont. USPS Postmasters: send address changes to: Transactor, PO Box 338, Station C, Buffalo, NY, 14209.

Croftward Publishing Inc. is in no way connected with Commodore Business Machines Ltd. or Commodore Incorporated. Commodore and Commodore product names are registered trademarks of Commodore Inc.

Subscriptions: see order card at centre of magazine for complete ordering information and prices.

Send all subscriptions to: Transactor Subscriptions Dept. 10-85 West Wilmot Street, Richmond Hill, Ontario, Canada, L4B 1K7, (416) 764-5273. Note: Subscriptions are handled at this address ONLY. Subscriptions sent to our Buffalo address (above) will be forwarded to our Richmond Hill HQ. For best results, use the postage paid card at the centre of the magazine.

Quantity Orders: In Canada: Ingram Software Ltd., 141 Adesso Drive, Concord, Ontario, L4K 2W7, (416) 738-1700. In the USA: IPD (International Periodical Distributors), 11760-B Sorrento Valley Road, San Diego, California, 92121, (619) 481-5928; ask for Dave Buescher.

Editorial contributions: writer's guides are available by writing to the Richmond Hill address above. Write to the editorial department (same address) with article proposals or articles for consideration.

All material accepted becomes the property of Croftward Publishing Inc., except by special arrangement. All material is copyright by Croftward Publishing Inc. Reproduction in any form without permission is in violation of applicable laws. Write to the Richmond Hill address for a writer's guide.

The opinions expressed in contributed articles are not necessarily those of Croftward Publishing Inc. Although accuracy is a major objective, Croftward Publishing Inc. cannot assume liability for errors in articles or programs. Programs listed in Transactor for the Amiga, and/or appearing on Transactor for the Amiga disks, are copyright by Croftward Publishing Inc. and may not be duplicated or distributed without permission.

Transactor for the Amiga is produced using Amigas and the following products:

Professional Page
Deluxe Paint II
Transcript
Digi-View

Final output by Vellum Print & Graphic Services, Inc., Toronto

Printed in Canada by Bowne of Toronto, Inc.

AbsExecBase

Amiga Developers' Conference 1989

I just got back from DevCon '89 and my head is still spinning – and not just because of an over-indulgence in 'networking'.

No, my disorientation is happily due to the staggering scope of Commodore's plans for its Amiga line of products – the most amibitious program of development I have ever seen the company undertake.

System software 1.4 highlights:

- the inclusion of the AREXX programming language
- full support of the new graphics capabilities provided by the Enhanced Chip Set
- scalable fonts
- support for networking in the form of record locking, file links and notification
- Fast File System (which will now incorporate the old file system) in ROM
- a Workbench improved in both appearance and functionality
- a library that will provide, among other things, standard file and font requesters
- the conversion of all c: commands and much of the dos.library to C
- preparations for virtual memory
- the addition of new screen types and modes, and new gadgets to Intuition
- extensive improvements to keymaps, the timer device, the trackdisk device, preferences and layers

The new Agnes and Denise chips' features include:

- addressability of 1 Mb of CHIP RAM
- a 'SuperHires' mode that provides resolutions of 1280 x 200, 1280 x 400 interlaced, with four colours from a palette of 64
- a 'Productivity' mode that provides a resolution of 640 x 480 *non-interlaced*, with four colours from a palette of 64 (this mode requires a bisynchronous or multiscan monitor)
- enhanced 'genlocking' capabilities
- increase of blit size to 32 Kb x 32 Kb

When? No committment from CBM. The disks handed out to the developers were Alpha only. Considering the extensive nature of the company's plans, I would say that it's going to take at least a year before 1.4 will be available to the public.

The development of the Amiga 3000 was officially announced. All that Commodore would say publicly about the A3000 was that it would have a 68030 on the motherboard.

But the most surprising developments were in the sales and marketing area. For the first time, the president of Commodore USA addressed the delegates to the Conference. Harry Copperman, who had only been at the position for 40 days, introduced an array of strategies that include:

- the establishment of a VAR (Value Added Reseller) program
- funding from CBM to selected developers for product development and marketing
- the establishment of a program to target higher education
- an ambitious advertising campaign
- a redesign of the appearance of the Workbench and a new emphasis on user interface consistency across applications
- a commitment to increased spending on research and development

While the new software and hardware developments were exciting, the new sales and marketing plans are crucial to the long-term health of the Amiga. Without them, CBM cannot hope to reverse the downward slide of Amiga sales in the US. The new programs mean that CBM will have finally done what IBM, Apple and other computer companies have been doing for years. Here's hoping it's not too little, too late.

* * *

We're pleased to announce a new addition to the *Transactor* team: Paul Bosacki will assume the rôles of assistant editor on both *Transactor for the Amiga* and *Transactor Classic*. Paul began hacking a Commodore 64 three years ago; since then he has written articles describing his own 8-bit RAM expansion designs; his latest implements a 1-Mb C64. Paul brings a slightly different perspective to the editorial team. Says Paul: "Hardware's the thing; software's a necessary evil." Should make for some interesting reading!

Tim Grantham
Editor

M.A.S.T.
Technical Excellence

"The Matchbox Collection" A POWERHOUSE OF PERIPHERALS.

MASSIVE MASS STORAGE FOR THE AMIGA 500/1000/2000

Maybe you want the unrivalled FLEXIBILITY of TINY TIGER — our UNIQUE standalone SCSI system, compatible with all Amigas.

POWERUSERS may prefer FIREBALL for their Amiga 2000. This true DMA, DRIVE ON A CARD, is designed to burn the competition on **price** and **performance**.

All MAST hard drive systems utilize State Of The Art, 3.5" SCSI drive technology. Both TINY TIGER and FIREBALL include a BUDGET and PERFORMANCE PLUS line of drives.

The BUDGET range comprise Seagate 30 & 50 megabyte drives.

The PERFORMANCE PLUS range feature FUJITSU 3.5" SCSI mechanisms — an indication of the reliability you should expect from this product. Naturally as another quality MAST product, it is SMALL, LOW POWER and covered by our ONE YEAR limited warranty.

PERFORMANCE PLUS drives are available in 45, 90, 135, and 180 megabyte configurations. They are voice coil, fast, and feature their own 24k byte RAM buffer for even faster access. The 45 meg mechanism is SUPERSMALL — at only 1" high it is the same size as our miniscule UNIDRIVE floppy.

TINY TIGER™
THE PURR-FECT SCSI SYSTEM

IF YOU CAN PLUG IN A PRINTER
THEN YOU ALREADY KNOW HOW TO
INSTALL A TINY TIGER
— Simple as that!

★ includes SCSI interface, drive and power supply
—all in one neat little case

★ drive is preformatted with workbench 1.3 installed
—power on and away you go

★ plugs into A500/A1000 and A2000
—great if you upgrade and ideal for moving software from one computer to another

★ connects thru 8 bit printer port
—a natural for SCSI — its 8 bit as well

★ fast — sure is — try this:
copy all of workbench 1.3 from RAM: to RAM: — you can't get much faster. It should take 22.5 secs. To do the same from Tiny Tiger to RAM:=27.6 secs. IN REAL WORLD TESTS — TINY TIGER PERFORMS.
— at lowest price per megabyte

★ printer is passed thru — transparently

★ pass thru for additional SCSI devices

★ TINY TIGER is attached by flexible cable and can be located in a convenient position
— DOES NOT OBSTRUCT USE OF A500 KEYBOARD

★ Bus expansion is left free for other important peripherals
— like memory

FIREBALL™
HARD DRIVE ON A CARD
FOR THE
AMIGA 2000

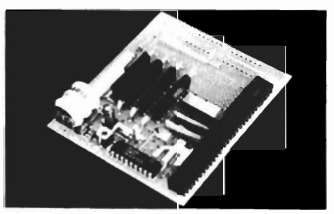
★ SCSI INTERFACE AND DRIVE
★ TRUE DMA
★ AUTOBOOT
★ VERY FAST
★ LOW COST
★ INDUSTRY STANDARD 50 pin CONNECTOR ON SCSI PASSTHRU

30 meg to 180 meg

CAPACITY	FIREBALL	T/TIGER
30 meg	$ 599	$ 629
50 meg	$ 669	$ 699
45 meg	$ 799	$ 799
90 meg	$1099	$1129
135 meg	$1369	$1399
180 meg	$1569	$1599

REMEMBER — T/T PLUGS
INTO ALL AMIGAS FOR A
CLEAN, UPGRADE PATH
— DON'T LIMIT YOURSELF
★ BUY TINY TIGER ★

MICROMEGS™ 512 K RAM AND BATTERY BACKED CLOCK



A500 ONLY
$169

This alternative to the A501 is only half the size and uses a fraction of the power — important for power conscious A500 owners.

THE ULTIMATE EXTERNAL FLOPPY DRIVES

UNIDRIVE

★ SUPER SLIM LINE
★ VERY LOW POWER
★ NO CLICKING
★ MAY BE SWITCH DISABLED
(TWIN DRIVE HAS TWO SWITCHES)
★ QUALITY FUJITSU MECHANISM
★ POWERED FROM AMIGA™ OR OPTIONALLY AVAILABLE PLUGPAK

UNIDRIVE $159

TWINDRIVE

★ 1 YEAR LIMITED WARRANTY
★ UPGRADE POLICY—BUY A UNIDRIVE AND HAVE IT FACTORY UPGRADED TO A TWINDRIVE ANYTIME
★ REMOVABLE CABLE
★ ROBUST BEIGE METAL CASING
★ HINGED DUST COVER
★ QUIET OPERATION

TWINDRIVE $299

MEMORY TO GO!
NEED MEMORY FOR YOUR A500/A1000
TRY THIS — MINIMEGS™
2 MEGABYTES OF EXTERNAL RAM
in a very small package

NEW LOW PRICES

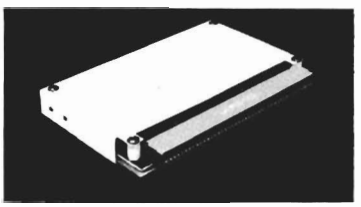
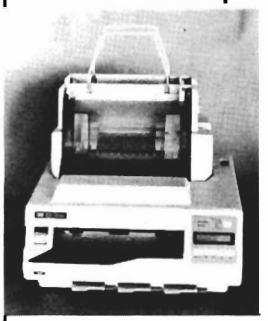


PHOTO OF A500 VERSION

- Auto Configure
- Low Power
- 1 Meg DRAM's
- RAM Tachometer
- Auto Configure LED
- Very Small Size
- Zero Wait States
- Compatible with A501

512K $329 **1 MEG $489**
2 MEG $779 (add $10 for all A1000 versions)

M.A.S.Terprint™ $1695



300 D.P.I.
DESKTOP PAGE
PRINTER

- FUJITSU ENGINE
- LED TECHNOLOGY
  — for increased
  — reliability
- SMALL FOOTPRINT
  —only 16½"x16½"
- DUAL PAPER FEEDER
- HP-PLUS® COMPATIBLE

M.A.S.Terprint PS™ $3495

This is the POSTSCRIPT® version of the page printer. It has 35 fonts and 2 meg of RAM (optional 4 meg). Additional POSTSCRIPT font cartridges can be plugged into the printer.

IMPORTANT: MASTerprint can be upgraded to MASTerprint PS anytime — lets you save up for POSTSCRIPT.

AMIGA 2000 OWNERS
OUR NO-CLICK INTERNAL
DRIVE IS A GREAT
DEAL AT $109

MAST PERFORMANCE PLUS HARD DRIVES AND ALL OUR FLOPPY DRIVES FEATURE FUJITSU MECHANISMS

FROM JAPAN'S LEADING COMPUTER COMPANY

MEMORY & STORAGE TECHNOLOGY INC.

M.A.S.T. U.S.A.
3881-E Benatar Way
Chico, CA 95928
(916) 342-6278 Tel.
(916) 342-6281 FAX

M.A.S.T. U.K.
Unit 1, Dippen
Brodick, Arran
Scotland, KA278RN
(077082) 234

M.A.S.T. AUSTRALIA
178 Pacific Hwy.
St Leonards NSW 2065
(02) 436-2600 Tel.
(02) 437-4919 FAX

Amiga, A500, A501, A1000, A2000, Fujitsu HP+ and Postscript are trademarks of their respective companies. All prices and specifications subject to change without notice.

Amiga Dispatches

A bolt of lightning...

by Donald A. Curtis
Copyright (c) 1989 Donald A. Curtis

Don Curtis is a police officer in Denver, Colorado, who has for the last three years been assigned to program design and development along with system design and maintenance of ten AT&T Unix 3B2 computers. In his spare time, Don is an assistant sysop on CompuServe's Amiga forums.

I was sitting around this afternoon trying to decide what I wanted to discuss this time, when an idea hit me like a bolt of lightning! That was mainly because that is exactly what it was... a bolt of lightning outside. Yes, this is the thunder and lightning season. Computers and static electricity don't mix very well, and since lightning is nothing but an awful lot of static electricity, computers and lightning don't mix at all.

Let's start out with what can occur if you computer gets a jolt of Electro Static Discharge (ESD). In its most common form, ESD makes itself known when you walk across a room and touch a metallic object and get a spark. We're surrounded by static electricity all the time – you really can't avoid it entirely, but you can take precautions to minimize it.

When that static mixes with your Amiga, you can have some fairly serious problems. As a general rule, solid state devices, like computer chips, either work or don't work - there's not much ground in the middle where they 'sort of work'. Or at least, that's what I've always been taught. However, after a recent discussion in the CompuServe Amiga Tech Forum, I learned that ESD can cause some very strange things to occur. Integrated circuits (ICs), consist of tiny transistors, capacitors and resistors all hooked together by tiny traces on the surface of the chip. As the number of these discrete devices on a chip has gone up, their size has gone down. As their size goes down, their ability to withstand overloads drops also. That's because a circuit trace can only carry so much electricity, or so many electrons. When you try to shove too many electrons down a path all at once, the pathway heats up and literally melts.

If the circuit completely melts, it can open up, and the circuit no longer works. However, other things can occur besides melting. The circuit can crack. It seems to work fine, until it heats up and then the crack widens to the point where contact is no longer being made. Again, the circuit will no longer work. The circuit can melt partially. That would leave an even smaller path for the electrons to go through. That can cause some really bizarre things to occur. A specific circuit in an IC may work fine at low current levels (small number of electrons) but at higher current levels (large number of electrons) it may break down and not work. Other weird things can happen: pieces of melted circuit can form a bridge across other circuit paths. Holes can be poked in one of the layers of semiconductor material that makes up an IC.

Get the idea? Intermittents can occur, or a specific piece of an IC may not work, while the rest of it seems to work fine. Since the Amiga, like other computers, uses what are called Very Large Scale Integrated (VLSI) circuit chips, each chip does several jobs. The two 8520 chips in the Amiga are an example of this. The 8520 is a custom chip made by Commodore, it's a 'Peripheral Interface Adapter' or PIA chip. Each one has several jobs to do. One handles floppy disk input, while the other handles floppy disk output. One handles the parallel port, while the other handles the control lines for the serial port. They do several other things also.

Let's take a look at the one that handles the parallel port (U10 in the A2000). It has a separate line for each bit going to, or coming from the parallel port. Eight lines, or bits, make one byte of data. Now, let's say your parallel port got zapped by static. The discharge went inside the 8520 and weakened the circuit that generates the least significant bit (or bit 0) of the parallel port. Try printing a letter and you'll find that half the letters will print just fine, the other half will print only some of the time, and the rest of the time, you'll get the wrong character printed.

And now you know why a blown 8520 (a relatively common problem) can cause such erratic behavior. The whole chip isn't damaged, just a piece of it – perhaps only a small piece.

So that brings us to the question, 'What can I do to prevent it?' The first thing should be obvious: turn the Amiga off before you plug anything into (or remove) it. That's only a start though. As you know, you don't have to turn a light on to get a spark to

jump if you've got a static charge built up. Just getting near the wall plate can cause a spark to jump. The same is true for your computer. Just getting near a port when you've got a charge built up can cause problems. The answer to that is to ground yourself before you work on the computer. Again, that includes things as simple as moving the mouse plug. You ground yourself by touching a metal pipe, metal wall plate, or any other grounded metal object.

That's not always convenient, so there's a third option. Check at a computer dealer for a ground mat for your work area. This is a small pad that fits under the computer, will conduct electricity, and has a small wire that you can attach to a grounded outlet. This is probably the safest option as long as the mat is properly grounded. With one of those mats, every time you go near the computer you touch the mat and clear your static charge.

There are a few more options you can use in addition to the above. If you've got a rug in your work area, spray it with any of the commonly available static preventive sprays. They're intended for clothes, but work fine on rugs too. If you ever open up your Amiga, make sure you use a static discharge strap. These slip over your wrist and have an alligator clip attached by a long wire. You attach the alligator clip to the chassis of your Amiga. This puts both you, and the Amiga at the same electrical potential so there's no static buildup. Static sensitive parts come in a special wrapping to prevent buildup. Keep it, and if you ever have to remove that part (or any other) use that wrapping to prevent a charge from building up. Avoid touching the pins of any chip, especially if it's out of its socket. Some chips are so sensitive when out of a socket that static charges too small to feel can destroy them.

And above all else, make sure you know what you're doing. I know I've said this before, but it really is important. You've got a thousand or more dollars sitting on your computer desk; one wrong move, and it can become a pile of plastic and chips with no useful purpose until it's repaired.

Now the big stuff...

Ok, that takes care of the little stuff, normal buildups of static that occur every day. But what about the big stuff? You know, those big flashes of light in the sky followed by loud rumblings?

Lightning is exactly the same stuff that jumps from your finger to the wall plate at home, it's just a whole lot more of it. While it's hard to get a spark to jump an inch at home, lightning can easily jump a mile. Home static can cause ESD damage in a chip; lightning will just blow the whole computer up into a pile of burnt scrap, along with anything else nearby.

The normal precautions won't do you a bit of good. If you think that turning your computer off during a lightning storm will prevent any damage, think of the size of the power switch. A bolt of lightning that just traveled from a cloud into your home and up the wire to your computer certainly isn't going to worry about a tiny little gap in a power switch. It's just going to jump that as if it wasn't even there. Not only that, but many power supplies only switch the secondary off (the low voltage side of your power supply); the primary (the side hooked to your power line) is always connected.

You say you've got good surge protection devices on your line? Guess again... that little spark gap or MOV in a surge protector isn't going to drain off enough power from a lightning bolt to make a whit of difference. Lightning is hotter than the surface of the Sun; that's why it can take a tall stately oak tree and turn it into two thin pieces of wood in nothing flat. It instantly vaporizes the moisture in the tree and causes it to explode. Your surge is prone to a similar effect.

I'm not saying don't use a surge protector, in fact, I suggest that you do; it's just that some things are beyond their capabilities, and lightning is one of them. The only safe thing to do when a lightning storm is nearby is to unplug your Amiga. That includes anything that's attached to the Amiga also. Printers, plotters, monitors, modems, external drives, etc. Anything that's plugged into the computer can be an entry for a lightning bolt.

And, don't forget to unplug the phone line from the modem! Lightning isn't picky, it can come down phone lines just as easily as it can come down a power line.

That's about as safe as you'll get. Lightning is a tricky beast; it's been known to come in windows, jump over furniture, go out another window and all sorts of crazy things. If it wants to eat your computer, it will one way or another. You can only do so much.

Your local power company has devices on the power lines that attempt to bleed off lightning strikes – that helps. If you live in an area where the utilities are underground, that helps also. There's less chance of a lightning strike on a power line; but again, nothing is foolproof. Even with all the best power line devices, underground wires, properly grounded house wiring and everything else, it's not unheard of for lightning to strike a house and enter the house wiring in the attic or anywhere else it wants to.

I've got my equipment on a couple of surge protected power strips, and when lightning is near (a few miles or so), I unplug the strips and the phone line from the wall and wait it out. No, I've never had my house hit by lightning, never had a TV or radio or any appliance blown up by lightning, but I'm not about to take a chance. I've got too much invested in my Amiga to take the chance I'll never get a lightning strike. We've got some high tension lines at the end of my block, and I've seen them get hit, so I've got no doubts that I could also.

How close is too close for lightning? That's hard to answer. Those high tension lines run to a sub-station a few blocks away. From that sub-station, the voltage is cut to distribution levels, then to a pole transformer and then to my house. When that high tension line got hit, nothing happened at my house except the lights flickered once. The equipment at the sub-station took care

of the lightning. So even a half a block away was okay in that particular case. Well, my rules are simple. If I can hear the thunder, I turn my computer off. I then watch the sky and when I see a lightning bolt, I start to count. If I don't get to 15 before I hear the thunder, I unplug the equipment. That means that when a storm is within about five miles, I quit working, and when the storm gets to within about three miles, I unplug it.

Now to the lighter side...

I get questions every now and then about what the flashing LED on the keyboard means during startup, well, here's the rundown:

- One flash means the keyboard ROM failed
- Two flashes means the keyboard RAM failed
- Three flashes means the 'watchdog' timer failed
- Four flashes means a short exists between two row lines or one of the seven special keys.

These flashes come in bursts with a short interval between them. Don't forget, the LED comes on briefly during self test; if it goes off and stays off, then all is well. You only need to become concerned if the boot fails and you see the LED flashing one of the above sequences. If you get reoccurring failures, it's time for a trip to the repair shop.

During startup, the screen colours are cycled and if an error occurs, cycling will stop with the screen colour giving a clue as to what's wrong:

Dark Grey screen	Hardware is okay
Light Grey screen	Software is okay

That's all you'd normally see, but additionally you could see the following colours:

Red Screen	Bad ROM checksum
Green Screen	Bad RAM
Blue Screen	Bad custom chip
Yellow Screen	CPU exception

If your screen stops on any of those four colors, it's time for a trip to the repair shop. If all you see are the grey screens, you're in good shape.

Panasonic has announced an 11 Meg 3.5" *floppy*! The Panasonic 3511 comes equipped with an embedded SCSI controller that includes a self-formatting function. It uses a special metal disk and micro stepping motor to get 640 tracks on the special disks. The controller is also capable of read only operations on a standard 3.5" floppy disk. Average access time is reported to be 60ms and track to track time is reported to be less than 10ms. Production quantities are scheduled to be available by July of this year and the list price is expected to be $250.

Spectrum HoloByte expects to have an updated version of *Falcon* available by August. This updated version should have no problems working on a 68020 (or '030) equipped Amiga. The software will include a script that allows you to put the software on a hard disk. Upgrade price will reportedly be $2.50 if you send them your original disk, or $7.50 if you call and have the upgrade sent via mail. To obtain the upgrade by mail, you must already be a registered user.

Some folks are getting upset with Commodore over the A2090(a) and problems with it. Rumour (and that's all it is right now) has it that Commodore is going to bring out a third form of the A2090 and call it the A2091. This half-card reportedly will fix all the problems with the current versions of the A2090. I have no information on whether this supposed A2091 will be a second card, or a replacement for the A2090(a). What the folks who are upset would like is a ROM fix for us current owners of the A2090 and Commodore's promised autoboot card for those of us that have the old A2090. They're saying, and maybe rightly so, that upgrading from the A2090 to the A2090a to this supposed A2091 is a pain. If all these upgrades are really needed due to defects in the original boards, then shouldn't Commodore make the current owners some price reduced offer to upgrade? Sounds reasonable to me, but then I don't have all the facts. I'll let you know more as soon as I know more.

Version 1.4 of the Amiga Operating System is on the horizon. As it stands right now, many of the current software packages will 'break' when used with v1.4. As v1.4 gets nearer, exact details will be know about what will break, and what won't... but for right now, one of the main concerns is that v1.4 will allow different system fonts. Any program that makes an assumption about what the system font is may very well break with v1.4. Other programs that make assumptions about maximum screen widths and heights, or about fast RAM being available may also go down the tubes. In fact, any program that makes any assumptions about the machine, other than those supported by the 'rules' may break on any OS change.

There's a problem with certain A2000 motherboards and accelerator cards such as the A2620. If you had your A2620 installed by a dealer, the problem should have already been taken care of, but if you installed it, or if you have a different accelerator... read on. Some A2000's were built with a Signetics 74ALS245 bus driver chip at position U605. This chip isn't fast enough to pull a critical expansion bus signal up at the end of a DMA transfer. Changing the chip to another manufacturer's part will fix the problem, but there's an easier way: just add a 3.3k pullup resistor between pin 20 and pin 11 of the chip. Dealers are supposed to be aware of this fix and can handle it for you. The Signetics parts can be identified by the big 'S' on the chip. No other manufacturer's part has shown this problem.

One final note, we recently had an argument over whether the Amiga is a 'home' computer or a 'business' computer. Well, there's a very simple answer for that... it's both. If you're able to take the current software offerings and run a business on the Amiga, it's a 'business' computer – the hardware is certainly more than capable of the task. If you're perfectly happy to sit at home with an Amiga, and have fun with it, then it most certainly is a 'home' computer. The best of both worlds, in fact.

Program With The Best!

Instant Interfaces!

Power
Windows
Version 2.5b

PowerWindows enters its third year as a unique, indispensible tool used by the vast majority of Amiga software developers. If you program in Assembler, C, Modula-2, FORTH, or even BASIC, you can use PW to design user interfaces for your programs. You do the design and PW generates the code. Fair enough?

"If Inovatronics hadn't done PowerWindows and done it right, I would have had to do it myself. I especially like to examine other people's interfaces with the 'Grab-A-Window' feature."

-Rob Peck, Author of The Amiga Companion and Programmer's Guide to the Amiga.

only $89.95

The "Complete Assembler Programming Environment" continues its quest to be the best, with these important new features:

- >HLink-CAPE's new linker, custom designed for assembly language. Smaller and faster than competing linkers.
- >Profiler- quickly isolates any bottlenecks in your program's code.
- >Code Module Dismemberer- examines a section of a program, disassembling code and dumping data.

Another major improvement in 2.5 is SPEED! CAPE now assembles 70,000 lines/minute with bursts up to 100,000 lines/minute. Written by Wesley Howe.

only $89.95

The Optimum Assembler

C.A.P.E.68k

New!Version 2.5

"...amazing optimizations...CAPE68k leads Amiga assemblers into a new generation"

-Rhett Anderson, Compute's Amiga Resource Summer '89

INOVAtronics

The REAL POWER in Power Programming

8499 Greenville Avenue, Suite 209B
Dallas, Texas 75231 Voice:214/340-4991
FAX:214/340-8514 BBS:214/357-8511

PowerWindows, CAPE68k, and Inovatronics are trademarks of Inovatronics, Inc.
Amiga is a trademark of Commodore International

Watch for the next major development in Amiga software, coming this August from Inovatronics.

CanDo
Power Steering for the Amiga

ACCESS

The world of freely-distributable software

by Steve Ahlstrom
Copyright (c) 1989 Steven Ahlstrom

As the primary sysop on CompuServe's Amiga forums, and a veteran Amiga programmer, Steve Ahlstrom sees just about every notable public domain and shareware program that gets written, and in this column he tells you about the ones that have recently caught his eye. If you have some information you'd like to share through this column, you can leave a message to Steve on CompuServe (his PPN is 76703,2006), or send the info to us and we'll pass it on.

There seems to be quite a bit of confusion concerning shareware, freely distributable copyrighted software, and public domain software. The confusion seems to be among the software authors as well as the users: many of the redistribution clauses I have seen lately are contradictory. I hope to clear up misconceptions you might have regarding these classes of software.

Shareware

A shareware product is actually a commercial product with a non-standard distribution channel. The author asks for compensation for his work, usually in the form of currency.

The distribution of the program can take many forms. The author might release a fully functional program. The documentation usually states that you may 'test drive' the software. If you find the software meets your needs, you are requested to mail a check for the purchase price to the author. An example of this type of program is Access! by Keith Young.

Another common method, used to encourage payment for the software, is for the author to release a freely-distributable but only semi-functional demo of his program. If you want the fully functional version, you send payment to the author, who then sends you the complete version. Rick Stile's uEdit falls into this category.

The above distribution methods are the most common. There are some twists. In return for your payment, you might receive source code to the program, you might be put on a mailing list to be notified of future updates, or you might even be mailed updates as they become available. Some shareware authors request payment in more idiosyncratic forms: donations to non-profit organizations; or in goods (ie. diskettes) rather than currency.

Regardless of the distribution or payment channels, shareware programs are not free. They are copyrighted works and must be paid for.

As with more traditional software publishers, support given by the shareware publishers can range from excellent to poor. The advantage to shareware products is being allowed to use the program extensively to determine its suitability to your needs.

Freely distributable, copyrighted software

This class of software for the Amiga seems to be growing. This is software the author wishes to make available to anyone who wants to use it. No compensation is required. The author does claim ownership to the software and has the right to dictate how the software will be distributed.

There are many reasons why an author may chose this route. Many times the program may be the basis of a future commercial product. The author may want to share concepts but not give away the farm. This goes against the grain of the 'hacker ethic' which says that all knowledge and source code should be made freely available to anyone who wishes it. However, this concept is generally acceptable when a computer system is new. Everyone is trying their best to learn the intricacies of the system and they make their knowledge available to everyone else.

As the machine matures, there are more people programming the machine and techniques derived may become commercially significant. By not retaining control of the program, the author may find himself, at some later date, competing with another product using his own code. This type of work is often called 'FreeWare'.

Public domain

There are no restrictions on public domain software. Once someone has declared that their work is public domain no one can claim any rights of any type on the software. Anyone may use it for any purpose. It may be included in the code in other works without permission (because no one exists who has the right to grant or deny permission) or it may be sold as a product with no compensation to the original author.

Here is where the confusion lies. Many times I'll see a documentation file for a program that specifically states, "This is a public domain program". Then, later in the documentation, the author outlines the method of distribution. In these cases the author is being contradictory. By declaring his work to be public domain, he has given up all rights to it, including the method of distribution. He no longer has any control over the work. If the author wishes to retain some control over the work, he should attach a copyright notice (and register the work with the copyright office).

(Editor's note: In Canada, an author automatically has copyright upon creation of the work and cannot assign copyright to the public domain. The author also automatically retains 'moral rights': the work cannot be modified without the author's permission, even when he has assigned the copyright elsewhere, unless he has signed a moral rights waiver.)

Shareware ethics

Let's say you wanted to sell an item, say, a lawn mower. Your neighbor says he is interested, but wants to use it first to see if it is all you say it is. You would probably lend it to him. You fully expect your neighbor to either pay you for the lawn mower or return it. This is much the same with a shareware program. The author allows you to give his product a fair evaluation. If you use the program, the author has every right to expect payment.

While a lawn mower cannot be copied, computer software can be. The shareware author has made his product available for your evaluation. In most cases, the price of the product is much lower than a more conventionally marketed comparable product. There are no middlemen involved to increase the cost of the product. Traditionally, a distributor will take 10-15% of the suggested retail price and the retailer will get 10-40%, depending on whether the software is sold at a discount.

A shareware product can be priced at 55% less than a conventionally marketed product. In fact, shareware products are normally priced much lower. There is little or no distribution or packaging expenses, and for those products distributed via networks and BBSs, there are no media costs.

Shareware, if it meets your needs, is an incredible bargain. Pay for it. Most people would never consider shoplifting a product from a computer store; don't continue to use unpurchased shareware products.

LED by Khalid Aldoseri

Not too long ago I was talking to Khalid in AmigaForums' conferencing area. I mentioned that it sure would be nice if there were a way you could tell when there was activity in RAM:. For example, when using ARC or ZOO, I almost always decompress files to RAM:. Sometimes large files can take a bit of time while decompressing. I get impatient and start to wonder if the machine has locked up. Khalid said, "Hold on a sec!" and uploaded *LED*.

LED will flicker the Amiga's power LED as data is being read from or written to any specified device. RAM: is not a device, so it does not work there, but ASDG's VD0: qualifies.

The program is a little disconcerting to use. We've all trained ourselves to expect to meet the Guru when we see the LED start to flash. After using *LED* for a month or so I still find myself reacting negatively when I see the LED flashing from the corner of my eye. But the program works as advertised. I no longer wonder if ARC or ZOO is still 'doing it's thing' or if I've locked up. Now I wonder if *LED* is running properly!

LED is a freely distributable, copyrighted work, and is available in CompuServe's AmigaTech Library 14 as LED.ARC.

LV by Khalid Aldoseri

Khalid explained that while he was playing with flashing the power LED he thought of another use for it. *LV* was born.

Have you rebooted your system only to find that your floppy or hard disk was not validated? You were probably too quick on the three fingered boot.

This used to be a major problem with floppies under 1.1 of the operating system. A write would be done; the LED on the floppy would turn off; an update would be done and the LED would come back on. If you rebooted before the update you'd invalidate the floppy. The Amiga took a lot of abuse from its detractors – it crashed all the time and trashed floppies.

Since 1.2 the floppy LED stays on until after the update is completed. But you can still be a bit too quick on warm boot. LV helps. Every time a write is done to a device, Amiga's power LED will dim. After the update is done it will return to normal. What this means is if the LED is dim, don't reboot!

LV is a freely distributable, copyrighted work, and is available in CompuServe's AmigaTech Library 14 as LV.ARC. Khalid Aldoseri may be contacted by sending CompuServe email to 75166,2531.

FixIntuition

Have you run across the Intuition bug that causes Gurus when you switch screens? This usually occurs when you have two interlaced screens and one non-interlaced screen. I get hit by this at least once a day.

This is a known bug and hopefully will be fixed in 1.4. Until then, there is a very small program to fix the problem called FixIntuition. Just run it from your startup-sequence, and these lock ups will go away! I'm not sure who the author is. The short readme file is just signed "Roger". Well Roger, thanks! You've solved a very annoying problem. FixIntuition is available in CompuServe's AmigaTech Library 17 as FIXSCR.ZOO.

Coming up...

There have been a lot of reports of an Access! v1.43. There is no such beast – at least, not yet. Keith Young, the author, has been spending time working on a couple of commercial projects. He was working on Access! in his spare time. Unfortunately, he experienced a hard disk crash that was totally unrecoverable. Some code was lost. The good news is that he expects to have a new version in the near future (considering lead times, it may be available by the time you read this). Keith does not want to reveal the new features the next version will have, but full ARexx support is a definite.

Khalid Aldoseri is working on a program called ScriptIt. Currently in the final development stages, it is a scripting language that allows you complete control over any program. ARexx is not required, nor is anything else other than the program. This is a real winner. Khalid has turned down at least two offers for it to be a mainstream commercial product. He intends for it to be freely distributable. Look for a full review in a future column.

Notice to Subscribers

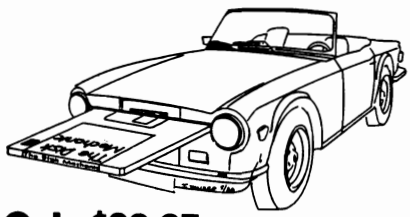
Renew early to avoid missing an issue! Check the expiry volume and issue on your mailing label, and make sure you renew your subscription a few issues in advance. For faster processing of your order, please pay by cheque or money order.

Notice to Non-Subscribers

Fill in the subscription card in the centre of the magazine and subscribe now! Get the most authoritative technical journal for the Amiga delivered right to your door.

The Disk Mechanic

Give your disks a Tune-up



Only $89.95

The Disk Mechanic is a comprehensive collection of utilities for Amiga DOS, including TuneUp, DoubleBack, DiskRepair, and WorkShop.

TuneUp: With TuneUp you can improve hard disk or floppy disk performance by up to 400%. Not a caching program, TuneUp is a sophisticated disk optimizer that actually re-organizes the information on your disk to improve system efficiency. TuneUp is fully compatible with Amiga DOS.

DoubleBack: DoubleBack is a high speed archival hard disk back-up program. DoubleBack supports multiple source and destination drives, file selection by name, date, protection, directory, and file note. Back up a full 20 megabytes in under 40 minutes. Selectively copy files between hard disks and partitions. Protect your data!

Sample TuneUp benchmark of some standard 3.5" disks:

Volume Name	TuneUp	BEFORE	AFTER	IMPROVED
Workbench 1.3	3:30	4.88	2.70	79%
Digi!	3:45	9.44	5.22	88%
WordPerfect	4:42	12.36	5.37	130%
SuperBase	3:37	19.39	9.66	100%
data	1:43	14.30	4.68	205%
archives	3:48	28.43	9.52	299%

* Times given are the average of three runs on a standard A2000 internal floppy drive. The TuneUp time is the time required to process the disk (in minutes and seconds). The BEFORE and AFTER times are the time (in seconds) required to get a directory of the disk. The more use a disk has had, the better the improvement!

Demo disk Available; call for details.

DiskRepair: With DiskRepair it is possible to recover accidently deleted or discarded files, salvage files from damaged disks, and reconstruct a corrupted hard disk. DiskRepair works with hard and floppy disks, and is compatible with the Fast Filing System.

WorkShop: For the expert user, the disk WorkShop is a full featured disk file and block editor. Examine and modify disk files, search for viruses, and change menus or requesters in popular programs.

The Disk Mechanic also includes a dozen small CLI based utilities to make managing your disks easier. We are now shipping version 2.5 with full support for Amiga DOS 1.3 and the Fast Filing System. The Disk Mechanic is not copy protected.

Dealer inquiries welcome
Amiga is a trademark of Commodore Amiga Inc.
WordPerfect is a trademark of WordPerfect Corp.
Digi! is a trademark of Aegis Development
SuperBase is a trademark of Precision Software, Ltd.

Lake Forest Logic Inc.
28101 E Ballard Road
Lake Forest, IL 60045
(312) 816-6666

MasterCard VISA

More than just a Hard Disk Backup Utility

ExpressCopy

Rapidly copies directories and files to floppy disk

FAST Copies directories and files from Hard Disk to Floppy Disk at up to 1 MegaByte per minute. Can format, verify, AND fill a new floppy disk with files in less time than it takes AmigaDOS to format a new disk!

SAFE Other Hard Disk Backup Programs this fast create NON standard disks that can only be used by their program. NOT ExpressCopy! ExpressCopy creates STD DOS disks that look just as if you had done a copy from your Hard Disk to floppy. If your Hard Disk failed, the backup disks can be used NORMALLY! This gives you a SAFE and EASY way to accesss important files you backed up.

Select files by their DateStamp, pattern matching, Archive Bit, and by source directory. All file attributes (DateStamp, Protection Bits, and FileComment) are retained. Options for setting the Archive bit for incremental backups, verifying the data written to floppy disk, and estimating the number of disks needed for the backup. Up to 4 copies of the backup disks can be created at a time, or disks can be pre-loaded in up to 4 disk drives. New disks are automatcially formatted and verified. Easy recovery if a bad diskette is found. Parameters to be used for backups can be saved in configuration files. Either Normal or Fast File compatible disks can be written. Specifically designed for effective multi-tasking. Backup restoration can be done using any file copy program, your favorite Directory Utility, or by ExpressCopy's Restore program.

ExpressCopy has no copy protection and can be used from both the CLI and Workbench.

Fully documented with a 65 page manual which includes a 25 page section with help and ideas on how you can better organize and manage your Hard Disk.

All these features and speed for ONLY:

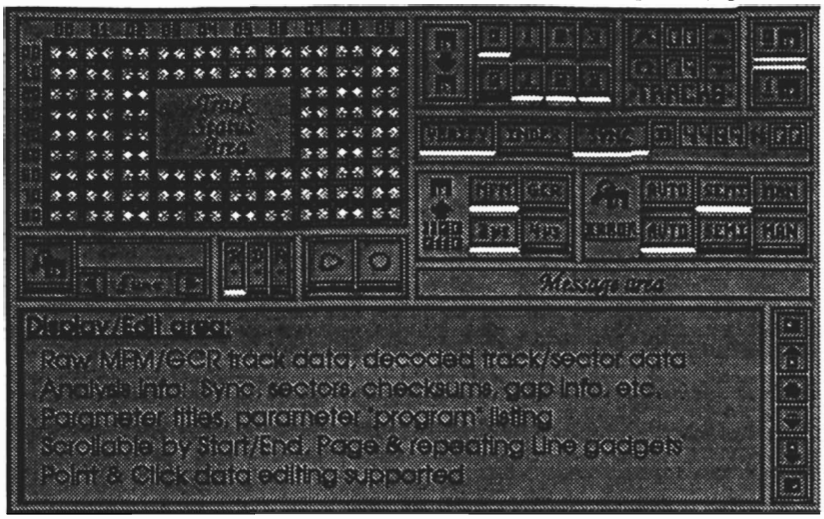
$44.95 US

Express-Way Software, Inc.
PO Box 10290
Columbia, MO 65205-4005
(314) 474-2984

Requires an Amiga with at least 512k ram and Version 1.2 or higher of AmigaDOS
Amiga, AmigaDOS, and Workbench are trademarks of Commodore-Amiga, Inc.

THE disk copier with *Public Domain* TEXT parameter files!

Aren't you tired of paying to upgrade parameter files?



The Reaper

FEATURES

- Totally User Programmable
- ASCII Text "Parameter" files
- Find shifted & scrambled headers
- Find Sectors by: min./max. syncs, min./max. any sync, headers
- Deprotect code wheel, and manual copy protection schemes
- Edit MFM/GCR/decoded data.
- Install on Hard Disk - not protected
- Execute custom programs from parameter files
- Assembly language WHERE IT COUNTS!

Unique "programmable" parameter file:

```
*AmigaDOS (parameter name)
0-79:sync on normal (s<9,s>12 any_sync)
```

Means: for tracks 0-79 do: turn Sync on, try normal track format, (if sectors <9 or >12, try ANY sync).
Totally User programmable ASCII parameter file.

(The Reaper Available 3rd qtr '89)

The Reaper.....39.95

Software from Hell.... 69.95

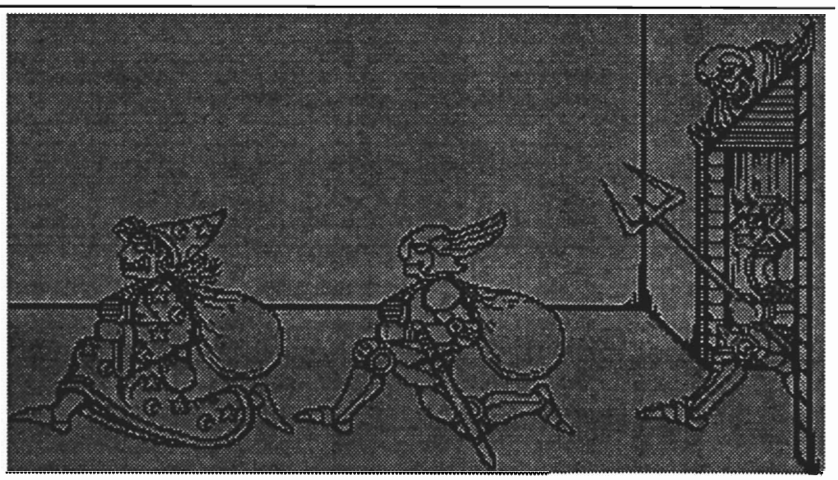
California Residents: Please add appropriate sales tax

Software from Hell...

Routines & Utilities for the programmer

AUDIO

open_audio	close_audio
open_channel	open_stereo
open_anychan	getchannel
close_channel	PlayNote
PlayFreq	PlaySamp
start_channel	stop_channel
reset_channel	flush_channel
donote	dofreq
dosamp	control_channel
lock_channel	unlock_channel
set_chpri	FreeNotes
MayGetNote	SetCycles
InitIOA	FreeIOA



Audio, Screen, Utilities - C source; Disk - Assembly. Get over that LONG learning curve and focus on writing the software you want to write!

DISK

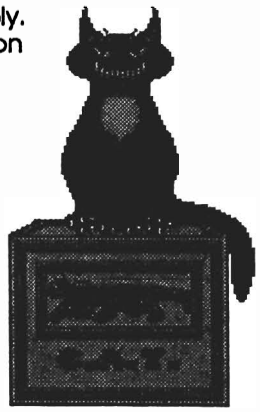
SetBusy	ClrBusy
MotorOn	MotorOff
HeadSelect	HeadIn
HeadHome	HeadOut
HeadPos	indexwait
ReadTrack	WriteTrack
TestProtect	DiskInDrive
CountTrackBytes	
TimeDelay	

SCREEN

init_screen	read_screen
read_background	read_palette
ReadCMAP	FindIFF
IFFRead	ReadBrush
load_menu	load_shape
put_shape	free_shape
rprintf	gprintf
BlitCopy	BlitShape
FreeShapeList	...and more...

UTILITIES

atags
stags
copyfrom
copytil
man
showincs
sweep
sync
xtabs



Conceptually Advanced Technologies

P.O. Box 3302, Santa Monica, CA 90403 **(213) 452-1732**

Mastering The Raster

Deluxe Paint III: civilized artists use Silva-ware

by Derek Grime

Derek Grime is president of Beyond Graphics, a Toronto-based studio that uses the Amiga for a variety of graphics, animation, and video applications. In this regular column, Derek talks about using the Amiga for professional work. Derek can be contacted at Beyond Graphics, P.O. Box 1249, Station F, Toronto, Ontario, Canada, M4Y 2V8.

The Amiga was not always the success story it is today. In the early days it was faced with extinction many times. I believe there are two reasons why the Amiga is still with us. The first was the support of the true Amiga fanatics. They knew the machine was superior to all the competition and stood by it through the bad times. The second factor was the high quality of some of the early software, specifically *Deluxe Paint*. Before we look at Dpaint in detail, let's take a trip back to the very beginning.

The Amiga burst onto the scene in the fall of 1985. The lavish world premiere was held in New York city. It was definitely a class act. The late Andy Warhol was among the many featured guests. Before an astonished crowd, he drew a portrait of rock vocalist Debbie Harry. Musicians played original compositions while dancers did interpretations of the scores. Animated performers on a bank of Amigas joined their flesh and blood counterparts on the stage. The press ate it up and christened the Amiga the machine of choice for the creative community.

A couple of weeks later, the video tape was screened at a meeting of the Toronto Pet Users Group. I was completely floored by the presentation and placed my order the next day. After a breathless wait, my Amiga arrived on Hallowe'en, nineteen eighty-five. I sat down to wait for the wondrous Amiga software to arrive. Two months later I was still waiting.

Circumstances changed quickly. By the beginning of 1986, the situation was starting to look pretty grim. Commodore was having financial problems, and the very future of the Amiga was in jeopardy. There was little quality software available for the Amiga, and what was out didn't have much power. Things couldn't get much worse.

Then Electronic Arts introduced Deluxe Paint, the program that opened the Amiga up to the artist. It was skilfully designed, and the user interface was a model of efficiency. There were numerous features that had never been seen on a micro before - many of the modes were not even possible on other PCs. Within weeks of its introduction the Amiga was being taken very seriously by computer artists.

To this day Dpaint remains the best-selling Amiga program by a massive margin. It is the standard by which all other Amiga productivity software is judged.

Electronic Arts continues to upgrade and refine the original package. Programmer Dan Silva has recently offered us the latest and greatest version of Deluxe Paint. The third major revision offers a score of new features. In some ways, Dpaint III took the Amiga marketplace by surprise. While rumour had it that Dpaint III would offer us hold and modify (HAM) mode, Dan Silva cleverly skirted the issue and presented us with powerful animation tools. The queen of Amiga programs would not stoop to joining the HAM paint wars - for the present time, anyway.

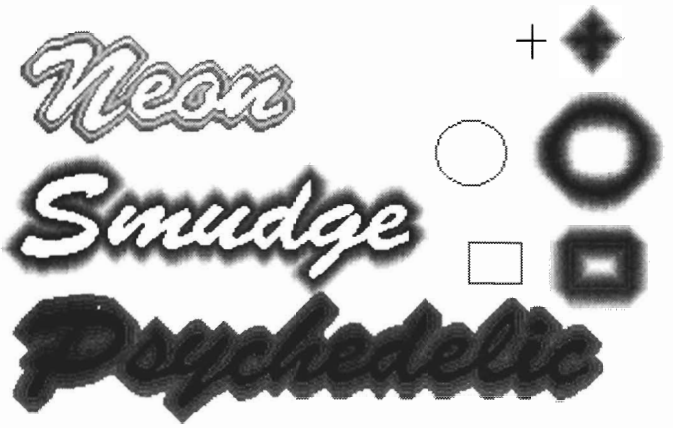
Let's look at some of these amazing new features, and how they can be used by you in your studio. For the most part, I'm going to ignore the major new features. They've been covered in detail elsewhere and many of them just bring Dpaint up to date. Features like better font requesters and overscan drawing modes have to be standard in any good Amiga program. There is so much more just hiding under the surface. After a couple months of extensive use, we may have stumbled upon some of the features that you might have missed.

A brief outline

Outline and trim are just two of the modes that can easily get lost in the throng of new toys. Located in the brush menu, they give you instant access to a range of new effects. Their most basic function is to add a one-pixel border or trim one pixel from the current brush. Run your copy of Dpaint III. If you don't own it, run down right now to your friendly neighborhood Amiga dealer and buy one.

Basic shapes take on a totally new look using outline. Using the range of greys that are in Dpaint's default palette, draw a

simple cross and pick it up as a brush. Outline once with each shade in the range. The end result appears to be made from stylized diamonds. Used as a pattern fill, this makes an effective background. All simple shapes can be given a whole new value using the outline tool over a range of colours.



Psychedelic lettering is an effect that we get a lot of mileage out of. While not for everyday use, the style is popular with bands and clients that are going after the youth market. The texture is jarring and even if it doesn't appeal to the average client, it certainly can't be ignored. Try consecutive outlines that step back and forth from dark to light. The words seem to shatter on the screen. Add a colour cycle, and you have an effect that will warp minds even across a crowded night club.

In fact, try cycling on the other examples as well. Smudge softly crawls into the background while neon appears to blink on and off.

Your name in lights

Now type in a hypothetical logo. The three lettering styles that are shown in the accompanying illustration are easy to obtain using outline.

Let's start by creating neon. Until now, neon styles have been difficult to render on the Amiga. When faced with a client's request for neon, we would draw all of the individual letters by hand or load in Aegis' Videotitler and use its built-in neon technique. DPIII now offers us another method. Pick up your logo as a brush and set yourself a range of three reds in the palette. Set the saturation range from light to dark. Select the dark red first as the foreground colour and press the small letter 'o'. This keyboard equivalent adds the first outline to the brush automatically. Now select the brighter red and outline again. Continue with the lightest red. When you get to the brightest red, reverse and go back to the darkest tone. Step through the colours again in reverse order, outlining with each. When you are finished, it should appear as though the original logo has a red chrome tube around the outside. Eliminating the center colour is easy. Fill the original logo with the background colour to leave just the newly created tube around the outside. Further enhancements can be added by just applying traces of white on the upper curves of each letter. Within minutes, you've created custom neon tubing that can be used in an illustration or for animated credits.

Smudge lettering is another simple style to obtain. Using the text brush, outline in a series of consecutive colours that fade into the background tone. When complete, smudge looks like chalk pastel and can be used in a variety of different ways. It's ideal when the client requests something that has a soft appearance. Computer graphics are known for their hard edges and technical coldness. Smudge puts a little warmth back into the process.

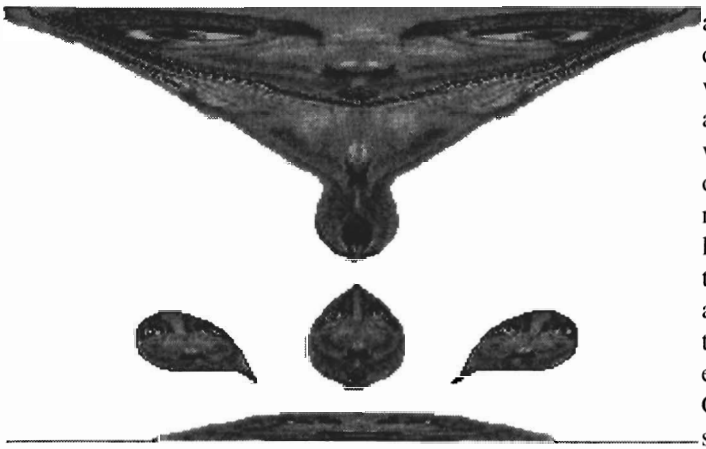
The flipside of outline is trim. Trim skins your brush down one layer, as if you were peeling an onion. Trim is useful when you want lettering to appear loose, like it was typed on a cheap typewriter or photocopied poorly.

Trim shines as an animation aid. A consecutive trim animated over twenty or more frames makes the illustration seem to dissipate as though it were made of sand. The trim dissolve is elegant and very different from the usual computer wipe effect.

Wrapping eloquence

A variety of new fill types have been added to Dpaint III. The best of the lot is Wrap mode. Wrap takes the current brush and mangles it into the current shape. There is no limit to the shapes you can fill. We've had a chance to use this before in both *Photon Paint* and *Deluxe Photolab*. Deluxe Paint goes a step further. Wrap and animation can be combined for added impact.

Try the following to see what I mean: Set up an animated sequence that's five frames in length - any series of sequential drawings will do. For the example pictured here, we used a five-drawing dripping movement. The large drop starts at the top of the frame and then splatters on the bottom. Not very exciting, I'll admit. Go to the scratch screen and load in a drawing that you've done previously. A

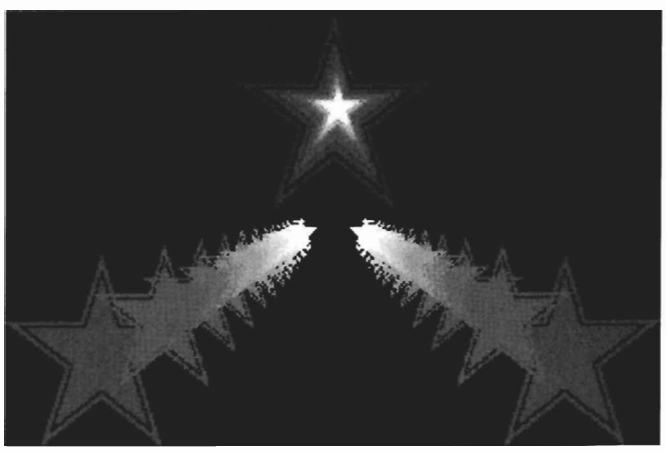


graphic of a face is very effective and there are some good ones available on the example disks that come with the program. Pick up the entire face as a brush, and turn on the Wrap mode in the fill requester. You can get there by clicking on 'fill' with the right mouse button. Rewind the animation to frame one. Fill in the first frame of the drip sequence and then move from frame to frame, filling as you go. On playback, the face will flow down the screen. We've done animated watches dripping in the style of Salvador Dali just for fun.

Imagine combining drawn shape animation with digitized images. Certainly all the frames still have to be drawn by hand, but think of the possibilities. Suppose you have a shot of the client's mascot or logo that fills the entire screen. Scan the image in with *Digi-View* and grab it as a brush. Using 'wrap', you could sequentially change the logo into anything you could picture, perhaps wrapping the first image into the shape of a new product they've just introduced. Many of you who do use your Amigas professionally in productions use them for wedding videos. Try taking a still picture of the bride and groom and animating them into the shape of a heart. It would make for a nice finish, and you only have to do the heart animation once. Afterwards, have the heart fly off into space. Your clients will never know how the effect was achieved, and neither will your competition.

Animation as a drawing tool

The big addition for revision III is animation. There is so much potential here that in the future I'll base an entire column on these powerful features. In the meantime, let's forget about animation for animation's sake and concentrate on the Move requester as a drawing tool. We first stumbled onto this when we were required to produce a sequence of a maple leaf flying into the distance. Using the automatic move completed the sequence in no time. The real work finished, we tried the same animation with the Trails function turned on. By accident we left the Cycle mode on as well. The results were amazing.



An object can be made to shrink into itself while leaving a multicolour trail in its wake. Try this on your Amiga: Draw a symmetrical shape using the first colour in the grey range in Dpaint's default palette. Circles and stars work well. Set the animation for ten frames. Pick up the shape as a brush and drop it in the center of frame number one (make sure that the brush 'handle' is in the center). Now turn on cycle mode (function key seven) and bring up the move requester. Set the 'Z' value to a high number like one thousand to shrink the brush, and then click on 'trails' to draw the picture for you. Preview the scene as an animation, but also view it as an individual graphic by going to the last frame. The last frame can be saved as a single picture if you like the results. Interesting variations on this technique abound. Try the same object again, only this time have it move diagonally across the screen while it's shrinking. Layers upon layers can be built up quickly, giving you textures that just aren't available using any other method.

Last but not least, I'd like to touch on the animbrush. The animbrush in Dpaint III is a series of objects that can be called up as one file. This means that a twelve-drawing animated walk cycle shows up in the program as one brush, but as you draw, the other frames appear. (The 'set' in Gold Disk's *Moviesetter* works with the same principle). It's meant to be used as a device to speed up the animation process, but nifty still graphics can be created easily. The 'pick up' option in the Animbrush submenu will capture as many pictures as you have animation frames. This requires massive amounts of RAM, but good results can be had with only a few frames. Imagine that you have an animbrush of a rotating head. By using the rectangle command, you can effortlessly create a border made of sequential pictures. Animbrush is more than just an animation tool.

As with any creative process, the ceiling is as high as you want it to be. Rules set by software are made to be broken. Seeing familiar tools in new and unconventional ways is the first step in freeing your ideas. Using Deluxe Paint III opens up worlds of possibilities to the dedicated Amiga artist. If you think illustrations done up to now have been impressive, you ain't seen nothing yet.

☆ ☆ ☆ ReSource ☆ ☆ ☆

Intelligent Interactive Disassembler for Amiga Programmers

"**ReSource** *makes full use of the Amiga windowing environment ... One of the features that distinguish* **ReSource** *is its ability to produce output which is compatible with most popular assemblers ... a bargain in this world of overpriced software." — Andre Marino*

ReSource is an interactive disassembler for the Amiga computer. Written entirely in assembly language, this is the most powerful disassembler available on *any* personal computer! Over 700 menu functions allow you to:

- ✔ Load from *any* file, read tracks, or disassemble memory directly.
- ✔ Save ALL, allowing you to continue disassembling later.
- ✔ Save to a ".asm" file, partial saves supported.
- ✔ Automated symbol creation:
  JSR -$1E(A6) becomes JSR _LVOOpen(A6)
  MOVE.L #$3EE,D0 becomes MOVE.L #MODE_NEWFILE,D0
  Over 5000 symbols supported!
- ✔ Incredibly smooth scrolling.
- ✔ Full key rebinding supported. Load/save keytables.
- ✔ Up to 38 user-defined macros available. Load/save/overlay macros, looping/conditionals supported, variable execution speed.
- ✔ Numeric and text processing functions allow you to automate many things that used to make disassembling so slow!
- ✔ Search function uses full ARP wildcards. Search forwards, backwards, or use the special bi-directional search!
- ✔ The *fastest* text rendering available on the Amiga.
- ✔ Forward and backward referencing allows you to quickly examine referenced code/data, and return to your original position. Nested references supported.
- ✔ Automatically/manually create labels. Manually create/edit/delete labels, symbols, end-of-line comments, full-line comments.
- ✔ Special support for "C" programs, including "A4" register data referencing.

US$75 **If you're serious about disassembling code, look no further!** **A$90**

To order **ReSource V3.05**
Send check or money order
Credit cards accepted

The Puzzle Factory
P.O. Box 986
Veneta, OR 97487
U.S.A.
(503) 935-3709

Glen McDiarmid
28 Marginson St.
Ipswich, QLD
AUSTRALIA 4305
(07) 812-2963

The worlds fastest programmers* use the new

JForth
Professional 2.0

from
Delta Research
Box 1051, San Rafael, CA, 94915, (415) 485-6867
Call or write for information.

JForth is an interactive programming language that compiles directly to 68000 machine code. With JForth, you can compile and interactively test your code in seconds. Our "Clone" feature can generate Royalty Free Applications as small as 3K by automatically removing unused code. Execution speed is comparable to 'C'.

Language Features

- 32 bit Forth '83 standard plus '79 & FIG,
- Compile from normal files or BLOCKs,
- Hashing for fast compilation, • Local Variables,
- Source Level Debugger with breakpoints,
- FFP single and IEEE double Floating Point,
- Disassembler plus RPN and Motorola Assemblers
- Object Oriented Dialect, source code included,
- Many utilities: Sort, File I/O, dump structures....

Amiga Support

- Call any Amiga library routine by name,
- Complete Amiga structure support plus precompiled include files,
- Easy toolboxes for Graphics & Menus,
- IFF support plus simple animation system with Wipes, Fades, Blits, etc,
- Extensive tutorials and many examples for beginners and experts alike.

* Mike Haas and Phil Burk used JForth to win the Worlds Fastest Programmer Contest, held in Anahiem, California, Nov. 1988.

Ask for JForth at a store near you. Suggested retail price = **$179.95**

Assembly Language, Part 6

Address modes, and the sizes of things

by Jim Butterfield
Copyright © 1989 Jim Butterfield

Jim Butterfield is a Toronto programmer and author whose articles, books, lectures and television programs have made his name a household word among Commodore users all over the world since the early days of the KIM-1 microcomputer.

Last time, we used an *algorithm* (a computational method) to convert from binary to ASCII, but it was limited in terms of the numbers it could handle. Now we'll do a more extensive version that will convert a full long word of unsigned binary data into a string; the number may be up to ten digits in length.

We'll take time out to discuss word sizes in more detail. As part of this, we'll look more closely at the pre-decrement and post-increment instructions. There will also be a chance to discuss the question of sizing both instructions and addresses.

The subject of address modes will also give us a chance to look at special address modes. In particular, we'll do a little string comparison using the special purpose compare instruction **cmpm** (compare memory).

The 68000 likes to be thought of as having 'orthogonal' addressing modes, which means that any instruction, in theory, would be able to use any addressing mode. But when you get down to details, there are many special-purpose addressing modes such as those used by **cmpm**.

We'll open our first window. Instead of using the traditional method, via the Intuition library, we'll use the quick CON: console window method. We won't be able to fill such a window with gadgets, menus, and mouse tracks; but it will give us a quick way to segregate our output.

Auto increment and decrement

A command such as **move.b #0,(a0)+** uses an addressing mode called *Address Register Indirect With Postincrement*. In this example, we would move the value zero into whatever address is given in register A0. Then we would add a value of one to A0, so that a similar instruction would then store to the following byte.

A command such as **move.b #0,-(a0)** uses *Address Register Indirect With Predecrement*. In the example, we would subtract a value of one from A0 *first*, and then store the byte into the resulting address.

Here's the clever thing: if we use other data sizes, such as **.w** or **.l**, the instruction will add or subtract a value higher than one. **move.w #0,(a0)+** will add *two* to A0 after the data has been moved, and **move.l #0,(a0)+** will add *four*. Be sure that A0 contains an even address value if you use these commands.

The PostIncrement and PreDecrement commands fit together nicely. They are especially useful if used together to put something temporarily to a stack. The stack could be one of your own, or it could be the program stack which is controlled by register A7. If you had a long word in D0 that you wished to put away for a short while, you could command **move.l d0,-(a7)**. At a later time, you must retrieve the value by **move.l (a7)+,d0**. Anything you put on the stack must be neatly cleaned off later.

By the way, A7 is often referred to as SP (for Stack Pointer), so that the above instructions might be written as **move.l d0,-(sp)** and **move.l (sp)+,d0**.

You may put words or long words onto the program stack. But remember that the stack pointer must always contain an even number, so although **move.b d0,-(sp)** will move a single byte onto the stack, the stack pointer will in fact be decremented by *two* bytes by this instruction. Similarly, if you grab space from the stack with the LINK instruction, be sure that you take an even number of bytes. Leaving the stack pointer with an odd address value will be fatal to your program.

Memory comparison with CMPM

Post-increment addressing works usefully when we wish to compare two areas of memory. The **cmpm** (Compare Memory) instruction uses two address registers, and bumps them both along as the comparison is made. For example: Suppose register A0 contains address hex 10000 and register A1 contains address 12000. If we were to command **cmpm.l (a0)+,(a1)+**, the four bytes of memory at 10000-10003 would be compared with those at 12000-12003; a following branch would be able to test for the results of the comparison, such as an equals condition. Meanwhile, A0 and A1 will be bumped along to 10004 and 12004. Another **cmpm** would continue the test at the next addresses.

This is especially convenient for comparing strings of characters in memory. We need only set two address registers to point at the two strings we wish to compare, and a repeated **cmpm.b** will walk along the strings, looking at each character in turn.

Note that the **cmpm** instruction accepts *only* post-increment addressing. You can't use pre-decrement to look from the back of a string, or any other address mode.

Instruction and address width

We understand that an instruction such as **move.l 4,a6** moves a 'long word' of data - four bytes - into register A6. To be more exact, the contents of 4, 5, 6, and 7 are delivered to the register. We say that the *width* of the instruction is *long word*. In a similar way **move.w** would move two bytes, and **move.b** would move a single byte only; in this last case, we would not be allowed to send the result to an address register.

If you don't give the width of an instruction, most assemblers default most instructions to .W, two bytes, and I wish they wouldn't. I would prefer that they insisted that you specify the width, since *word* is the least common width used in typical Amiga programming. We more often move full registers or addresses (with .L), or deal with characters (with .B).

Let's go back to the common instruction, **move.l 4,a6**. We know that the instruction width is long-word; but now, let's look at that first address.

Address 4 can be expressed as a long word, of course: hexadecimal 00000004 gives the whole thing in 32 bits. This makes the whole instruction, including this address, moderately lengthy; the address alone uses up four bytes. It seems like a waste of space for such a little address.

You are allowed to express such addresses using only 16 bits, two bytes. Hex 0004 looks more compact, takes up less space, and is exactly the same address. So you may give certain addresses (not too many on the Amiga) as 'word' addresses, and save space. Most assemblers will do this automatically, but some call for you to code **move.l 4.w,a6** so as to specify explicitly that the address, 4, is to be held as a word. Do not be confused when you see both .L and .W; the .L refers to the **move** operation, and the .W signals the size of the address. Four bytes of data are still moved.

Branch instructions can sometimes puzzle the beginner. An instruction such as **bpl.s** uses no data. The .S (for *short*) signals that the programmer believes that the branch is only a short hop, so that the instruction can hold the offset value as a small, one-byte, value. Roughly speaking, a short branch must specify a destination within about 128 bytes, forward or backward.

In some cases, your assembler is likely to detect a short branch and put it in automatically for you. This happens on backward branches, as would be used for loops. As the assembler ploughs through your program, it finds a line such as **bpl zonk**, and says: "I've already passed label *zonk*; I know where it is, and it's within Short range." In such a case, it would automatically switch the instruction to **bpl.s zonk**.

The assembler cannot make this kind of decision for forward branches. When it finds such branches, it has not found the label yet, so it can't guess whether or not a short branch would reach. It must make the conservative decision, and assume that the branch might be long. In this case, you must explicitly declare the branch as short if you believe it will reach.

Some assemblers - notably DevPac and Charlie Gibbs' A68K - are able to tell you if a forward branch could have been declared as short. If you get such a message, you can modify your source appropriately.

The project

Program *SeeDir* will allow a user to specify a file or directory. For example, **SeeDir df1:** would be a legitimate command. If a file is named, its size is given and that's all. If a directory is named, the program will go through that directory and report all files it finds there.

Well, almost all files. To give us a little experience in string matching, we will refuse to report any files that end with the letters ".info"; instead, we'll skip such files.

When we look at a directory, we'll open a new window (a CON: console window). In that window, we'll report any subdirectories we find in the drawer. If we find files, they will not show in the window; they will be reported in the standard output stream, which will normally be the CLI screen.

If we should give the command *SeeDir* with output redirection, the output stream will be switched. For example, **SeeDir >ram:xx df1:** will send its standard output to file *xx* rather than to the screen; but the CON: window will not be switched, since the program opens it and sends to it directly.

Let's go to the program and comment on coding details.

The program

I'll leave it up to you to add comments at the beginning of the program. Remember to put in the program name, the date you typed it, and perhaps a version number.

I'll start with the standard library calls; you might prefer to make these XREF definitions and let *BLink* match them up from your library files. The definitions might be brought in from an *include* file. You should be getting accustomed to your system by now, and be making decisions on how you want to arrange things.

```
; Exec library calls
_LVOAllocMem          equ -$C6
_LVOFreeMem           equ -$D2
_LVOSetSignal         equ -$132
_LVOCloseLibrary      equ -$19E
_LVOOpenLibrary       equ -$228
; DOS library calls
_LVOOpen              equ -$1E
_LVOClose             equ -$24
_LVOWrite             equ -$30
_LVOOutput            equ -$3C
_LVOLock              equ -$54
_LVOUnLock            equ -$5A
_LVOExamine           equ -$66
_LVOExNext            equ -$6C
;
MODE_OLDFILE          equ 1005
ACCESS_READ           equ -2
INFOSIZE              equ 260
; FileInfoBlock items
fib_FileName          equ 8
fib_EntryType         equ 120
fib_Size              equ 124
```

We open the program itself by asking the computer to give us a block of working storage from the stack; a 28-byte 'frame' obtained with the **link** command. Details on how we will use the frame are shown in the **EQU** statements below.

```
;
;-- Initial setup:
;        offsets from A5 frame
SysErr     equ -2        ; word, flags errors
tCount     equ -4        ; word, counts files
dosBase    equ -8        ; the DOS library pointer
argv       equ -12       ; Command line tail address
OutHandle  equ -16       ; 'normal' CLI output
String     equ -28       ; 12 bytes numeric work area
           link a5,#-28  ; 'global' work area frame
```

We have our data frame, and must remember to give it back when we're finished. Our next job is to set the number of files to zero (so far), and to stuff a zero byte behind our command tail string whose location is given in register A0 and whose length is defined in D0.

```
          moveq  #0,d1
          move.w d1,tCount(a5) ; file count =0
          move.b d1,-1(a0,d0)  ; c style string
```

There should be no spaces at the start of the Command Tail; but just in case, we'll skip past them. In fact, we'll go one location too far as we scan along. Note how pointer A0 moves ahead each time we exercise the next line.

```
spdump  move.b (a0)+,d0   ; slide over spaces
        cmp.b  #$20,d0
        beq.s  spdump
; note that A0 is +1 from start of string
```

The set-up from this point on should be fairly familiar to you by now.

```
; set up dos, output, here
  movem.l a0/a1/d0/d1,-(A7)
  lea     dosname(pc),a1    ; Name 'dos.library'.
  clr.l   d0                ;   Any version (0)
  move.l  $4,a6             ;   Using Exec library,
  jsr     _LVOOpenLibrary(a6) ; Open Dos library.
  move.l  d0,dosBase(a5)    ; Remember DosBase ptr.
  beq     StartupQuit       ;      dos not opened
  move.l  d0,a6             ; set Dos library
  jsr     _LVOOutput(a6)    ;   get CLI outhandle,
  move.l  d0,OutHandle(a5);   & then remember it.
  movem.l (a7)+,a0/a1/d0/d1
  move.b  -(a0),d0          ; no file name?
  beq     Prompt            ; then send Prompt data
```

We have corrected our Command Tail pointer, and may now save it. We will now call subroutine *GetDir* to do all the work for us. The file name (the Command Tail) may be placed on the stack, and *GetDir* will pick it off from there. We must remember to clean off this extra stack item later.

```
  move.l  a0,argv(a5)    ; save Cmd Tail pointer
; Using command tail file name, ask for dir search
  move.l  a0,-(a7)       ; file name pointer to stack
  bsr     GetDir
  move.l  (a7)+,a0       ; restore stack after call
```

GetDir has signalled any problems in variable *SysErr*, and has left a count of files in variable *tCount*.

```
; if SysErr is zero, wrap up .. else signal error
  move.w  SysErr(a5),d0
  bne.s   Ncase0
;  No errors. Test if no items found.
  move.w  tCount(a5),d0
  bne.s   GiveCount
  move.l  OutHandle(a5),d1  ; CLI handle
  lea     EmptyMsg(pc),a1   ; say 'Empty'
  move.l  a1,d2             ; output buffer
  moveq   #EmptyLen,d3      ; length
  jsr     _LVOWrite(A6)
```

```
    bra.s   endSwitch
; Found at least one item. Give item count.
```

The file count now in D0 is of length *word*. Before we convert it to ASCII, we'd better 'extend' it to a long word.

```
GiveCount:
;count is in lower word of d0. 'Extend' to full word.
    ext.l   d0
    bsr     MakeAscii          ; make a string of digits
    move.l  OutHandle(a5),d1
    lea     String+6(a5),a1    ; don't print 'em all
    moveq   #5,d3              ; only 5 digits
    move.l  a1,d2
    jsr     _LVOWrite(a6)
    move.l  OutHandle(a5),d1   ; CLI handle
    lea     CountMsg(pc),a1    ; say 'items'
    move.l  a1,d2              ; outbuffer
    moveq   #CountLen,d3       ; length
    jsr     _LVOWrite(A6)
    bra.s   endSwitch
```

If *GetDir* has returned an error signal, we should report it.

```
; Some kind of error flagged in d0. Analyze
Ncase0      cmp.w   #1,d0
            bne.s   Ncase1
; Error 1 (rare). Not enough memory
            move.l  OutHandle(a5),d1   ; CLI handle
            lea     NoMemMsg(pc),a1
            move.l  a1,d2              ; outbuffer
            moveq   #NoMemLen,d3       ; length
            jsr     _LVOWrite(A6)
endSwitch   bra.s   LastGasp
; Error 2 (you goofed). Cannot find file.
Ncase1      cmp.w   #2,d0
            bne.s   endSwitch      ; should never happen
            move.l  OutHandle(a5),d1   ; CLI handle
            lea     NoFileMsg(pc),a1
            move.l  a1,d2              ; outbuffer
            moveq   #NoFileLen,d3      ; length
            jsr     _LVOWrite(A6)
; No command tail or no file. Print Prompt/Info msg
Prompt      move.l  OutHandle(a5),d1
            lea     PromptMsg(pc),a1
            move.l  a1,d2
            moveq   #PromptLen,d3
            jsr     _LVOWrite(A6)
;**  }
LastGasp:   move.l  dosBase(a5),a1
            move.l  $4,a6          ; using Exec library,
            jsr     _LVOCloseLibrary(a6) ;close Dos lib
StartupQuit unlk    a5
            rts                    ; End of Program
```

Subroutine *GetDir* does most of the work, and is interesting in a number of ways. It has its own set of variables that it obtains as a stack frame; it gives these back to the stack before it returns, so that the calling program has no knowledge of them. Meanwhile, it also has access to the calling program's variables; and it takes the file name from the stack.

```
; Subroutine GETDIR ....
;   ..receives directory name from the stack;
;   ..prints directory file items;
;   ..returns tCount and SysErr values.
; Following are variables private to GETDIR
;            offsets from A4 frame
Lock        equ     -4
fibb        equ     -8
NameSave    equ     -12
ConHandle   equ     -16
;
```

You may recall that the calling program pushed the file name to the stack with a **move a0,-(a7)** command. At this point, *GetDir* cannot disturb the stack - its return address is on there - but it can 'peek out' the file name pointer.

A little calculation. The subroutine's return address is at 0(A7); it's four bytes long. Above that is the file name pointer, which will be at 4(A7). Note that we do not disturb the stack pointer; we just use it.

```
GetDir  move.l 4(A7),a1  ; grab file name from stack
        link   A4,#-16
        move.l a1,NameSave(a4)
; Allocate enough memory to hold FileInfoBlock
        moveq  #1,d0
        move.w d0,SysErr(a5)  ; Maybe Alloc error
        move.l $4,a6          ; this calls for Exec
        moveq  #0,d1          ; any type of memory
        move.l #INFOSIZE,d0   ; size of block needed
        jsr    _LVOAllocMem(a6)  ; go get it
        move.l dosBase(a5),a6 ; restore Dos Library
        move.l d0,fibb(a4)    ; save the block address
        beq    GDExit         ; quit if no memory
```

A few words about Locks. Locks for files and directories are called the same way, and the FileInfoBlock is brought in using the DOS call *Examine*. Until we reach that point, we don't know if the user has typed the name of a file or a directory.

The discovery of what kind of disk item we're dealing with (file or directory) changes our perception of what the lock has done. A file lock serves quite a different purpose from a directory lock, even though it's obtained the same way.

A *file* lock might be regarded as an 'interlock' that says: if any task is writing to this file, all other tasks must leave it alone. Several tasks can read the same file simultaneously, but writing to the file is a solitary job.

If you ask for a read lock on a file, you'll get it if nobody is writing to the file. If you get the read lock, nobody will be allowed to write to it until you release the lock.

A *directory* lock is quite a different matter. It does not keep any other task out (although it *does* prevent the directory from being deleted until the lock is released); instead, it 'opens' the directory so that you can see the contents (using DOS call *ExNext*).

```
; Get a lock on the requested file or directory
   moveq  #2,d0
   move.w d0,SysErr(a5)      ; Maybe File error
   move.l NameSave(a4),d1
   moveq  #ACCESS_READ,D2    ; get a read lock
   jsr    _LVOLock(a6)
   move.l d0,Lock(a4)        ; store the lock pointer
   beq    DropMem            ; couldn't get lock

; Get FileInfoBlock .. check whether file or dir
   move.l d0,d1              ; lock
   move.l fibb(a4),d2        ; file info block address
   jsr    _LVOExamine(a6)    ; go for the data
; Assume Examine never fails.
   moveq  #0,d0
   move.w d0,SysErr(a5)      ; no errors found
   move.l fibb(a4),a3        ; file info block address
   tst.l  fib_EntryType(a3)  ; file type positive?
   bpl.s  NotFile            ; positive, it's a DIR
```

If the user has specified a file, we'll set the count to one and print the file size and file name.

```
; User has named a file, not a directory
   addq.w #1,tCount(A5)
   bsr    PrintStuff
   bra    DropLock
```

On the other hand, if a directory is named, we have quite a bit of work to do. First, we'll open a console window to give ourselves a second output path.

```
; User has named a directory. Explore it.
;     Open console window:
NotFile    lea    ConName(pc),a1
           move.l a1,d1
           move.l #MODE_OLDFILE,d2
                        ; as if it's an INPUT file!
           jsr    _LVOOpen(a6)
           move.l d0,ConHandle(a4)
           beq    DropLock     ; if Open didn't work.
```

We supply DOS function *ExNext* with the directory lock and a place to put the FileInfoBlock. Each time we call, we will get a new FileInfoBlock telling us about another item within the directory. When there are no more items, we'll get a zero result from our *ExNext* call.

```
; Use this loop to explore the contents of a dir.
;    ExNext() will fail when there are no more items.
DirLoop:   move.l Lock(a4),d1   ; lock
           move.l fibb(a4),d2   ; file info block addr
           jsr    _LVOExNext(a6)  ; go for the data
           tst.l  d0
           beq.s  NoMoreDir
```

At this point, we have a valid item from within the specified directory, so we may add one to the file count. The FileInfoBlock will contain the item name and whether or not this item is a file or a subdirectory.

```
; Extract the name, count it, then test for sub-DIR
           move.l fibb(a4),a3
           lea    fib_FileName(a3),a2
           move.l a2,NameSave(a4)    ; log name
           addq.w #1,tCount(a5)      ; add one to count
           tst.l  fib_EntryType(a3)
           bpl.s  SubDir             ; positive if DIR type
```

When we reach this point, the item we have found is a file and not a subdirectory. We will test to see if the file name ends with ".info"; if it does, we'll skip the output part.

Instruction **cmpm** will be used to compare the name with a fixed ".info" string. Since we wish to check the end of the name, we'll need to scan through the string to find the zero byte at the end. Register A2 is used for the scan.

```
; It's a file. Check for '.info' ending.
           move.l NameSave(a4),a2
TryCharx   move.b (a2)+,d0 ;search for end-of-str
           bne.s  TryCharx
```

Register A2 now points to a location *following* the terminating zero byte. To inspect the last five characters of the string, we must back up six bytes.

An easy way to trim back register A2 would be to use **subq.l #6,a2**. I'd like to show you a different way to do the same job. **subq** can use immediate values up to 8; the following **lea** instruction doesn't have such a narrow limit. Can you see how it works?

```
           lea    -6(a2),a2       ; back off 6 positions
           lea    InfoMsg(pc),a1  ; set up '.info' match
```

As we enter the loop, A2 points at the last five characters of the file name, and A1 points at the fixed string ".info.". Notice the extra character at the end of this string, guaranteed not to match the zero of the file name.

```
InfCmp     cmpm.b (a1)+,(a2)+     ; does this char match?
           beq.s  InfCmp          ; yes, look further
```

We fall out of the above loop when we find a mismatch. If we have matched the entire ".info" string, we'll drop out when we compare the trailing '.' character against the zero byte that ends the file name.

That's easy to test by means of backing up register A2.

```
; We get here when '.info.' no longer matches.
; If at END of the file name, it's an .info file
     move.b -(a2),d0     ; look at mismatched char
     beq.s  NoWrite      ; it's end-of-string!
; not an .info file; print the data.
     bsr.s  PrintStuff
     bra.s  NoWrite
```

If we have found an entry that is not a file - in other words, a subdirectory - we arrive here. I've arbitrarily decided to print *dir* information in the CON: window.

A FileInfoBlock for a directory does not give us a file size. To find that, we'd need to search the subdirectory, and add up all the file sizes we find there. For the purposes of this program, we'll just print "DIR" and the file name.

```
; A subdirectory. Print DIR message to CON: window.
SubDir  move.l ConHandle(a4),d1
        lea    DirMsg(pc),a1
        move.l a1,d2
        moveq  #DirLen,d3
        jsr    _LVOWrite(A6)
        move.l ConHandle(a4),d1
        bsr.s  PrintName
```

To check for a user CTRL-C signal, we must call *SetSignal*. That's in the Exec library, so we must temporarily change our A6 library pointer. If CTRL-C is not indicated, we go back for more directory elements.

```
; Before we loop, check for Ctrl-C.
NoWrite:
     move.l a6,d7          ; save Dos Library pointer
     moveq  #0,d0
     move.l #$1000,d1
     move.l 4,a6
     jsr    _LVOSetSignal(a6)
     move.l d7,a6          ; restore Dos pointer
     and.l  #$1000,d0
     beq.s  DirLoop        ; no Ctl-C, loop.
```

I like to 'echo' a CTRL-C type of signal as a reminder that the program did not finish the job.

```
; print Ctrl-C signal
     move.l OutHandle(a5),d1
     lea    CtrlCMsg(pc),a1
     move.l a1,d2
     moveq  #CtrlCLen,d3
     jsr    _LVOWrite(A6)
```

We should close the console window when we've finished going through the directory. If the CON: window contains important information, we might need to introduce a pause so that the user can examine the information before the window disappears. A fixed delay, or a 'press any key' routine, would do the job.

```
; Directory search ended.  Close console window.
NoMoreDir  move.l ConHandle(a4),d1
           jsr    _LVOClose(a6)
; Unlock the user-specified file or directory.
DropLock   move.l dosBase(a5),a6   ; set Dos Library
           move.l Lock(a4),d1      ; take off the lock
           jsr    _LVOUnLock(a6)
; Give back the InfoBlock memory block.
DropMem    move.l $4,a6            ; set Exec library
           move.l fibb(a4),a1      ; mem block address
           move.l #INFOSIZE,d0     ; size of block
           jsr    _LVOFreeMem(a6)  ; release memory
           move.l dosBase(a5),a6   ; set Dos Library
; GetDir has done its job. Give back the frame, exit.
GDExit     unlk   A4
           rts
```

The following subroutine prints the file size as an ASCII string. It then falls into the following routine, which prints the file name.

```
; Print file size (convert number to Ascii)
PrintStuff:  move.l fib_Size(a3),d0  ; get size
             bsr.s  MakeAscii ; change to ASCII str
             move.l OutHandle(a5),d1
             lea    String(a5),a1
             moveq  #11,d3   ; print 11 characters
             move.l a1,d2
             jsr    _LVOWrite(a6)
             move.l OutHandle(a5),d1
                                ; set up CLI output
```

PrintName might be called directly, or might be a continuation of the above logic. It might be expected to send to the standard output stream, or it might be asked to send to the CON: window. Whatever handle it needs will be in register D1. We will save a copy of this handle for the second part of the print routine; it goes to the stack.

```
; Print file name to CLI or to CON: outhandle
PrintName  move.l d1,-(a7) ; save outhandle to stack
           move.l NameSave(a4),a2
           move.l a2,d2            ; start of name
```

We'll walk through the string to find the zero byte that signals the end of the name. We could use postincrement coding again here. But it's a little easier to use D3 as an *index*. Note the reference to **d3.w** - we index only on the lower 16 bits of D3.

```
; Scan to find length of name, then print
           moveq  #0,d3
TryChara   move.b 0(a2,d3.w),d0
           beq.s  FoundEnd
           addq.b #1,d3
           bra.s  TryChara
FoundEnd   jsr _LVOWrite(A6)
```

We bring back the output handle from the stack in order to send the NewLine character to the same destination.

```
; Print NewLine to same output stream
        move.l (a7)+,d1          ; bring back outhandle
        lea    MsgNewLine(pc),a1
        move.l a1,d2
        moveq  #NewLineLen,d3
        jsr    _LVOWrite(A6)
        rts
```

Here is our expanded program to convert a full 32 bits to an ASCII string. An unsigned longword can generate a number as high as ten digits: 4294967295, to be exact. We will use a 12-byte area, allowing for a couple of spaces to follow the number.

```
; Subroutine:  Convert longword to ASCII string
;   --value received in D0;
;   --result placed in string at String(a5)
; First clear 12-character string to spaces.
```

The following code fills D1 with four ASCII 'space' characters. It then bangs the spaces, four at a time, into the string area, filling it with twelve spaces.

Such code is very fast: three **move.l** commands put twelve spaces into memory, which is much faster than the more 'natural' **move.b** within a twelve-count loop. Because of the speed, you'll see this kind of thing from time to time within the Amiga.

The code is included to show you this aspect of 68000 coding. Under normal circumstances, I would prefer the slower byte loop. The **move.l** method assumes that the area being initialized starts at an even address, and contains a number of bytes that divides evenly by four. Changing the size or location of buffer *String* would call for recoding of this section. **move.b** creates simpler and safer code.

```
MakeAscii  move.l #$20202020,d1 ; 4 space characters
           lea    String(a5),a1
           move.l d1,(a1)+
           move.l d1,(a1)+
           move.l d1,(a1)+
           lea    String(a5),a1
           moveq  #9,d2
```

As before, we're calculating digits from the low end of the number. Thus, if we had a value of 2468, we'd extract the 8 first and put it near the end of the buffer area. When we found the next digit, 6 in this case, we would 'back up' register d2 so that it would be stored ahead of the 8, and so on.

The code itself is lightly commented. In order to divide a 32-bit number by ten, we must do it in two chunks, dividing the high byte first and then working in the low byte with a second division. It's a little like the familiar long division we have all done.

```
; Calculate digits from low end of number.
DigLoop    swap   d0             ; high/low word swap
           moveq  #0,d1
           move.w d0,d1          ; high word to D1
```

The **divu** instruction is sufficiently slow that we'll take an extra instruction to skip over it if the value is zero. The high byte will often be zero.

```
           beq.s  Skipper        ; zero, don't divide
           divu   #10,d1         ; divide hi word
Skipper    move.w d1,d0          ; new hi word value
           swap   d0             ; restore hi/low order
           move.w d0,d1          ; set low value
           divu   #10,d1         ; continue division
           move.w d1,d0          ; new lo word value
           swap   d1             ; get remainder
           or.b   #$30,d1        ; make it Ascii
           move.b d1,0(a1,d2.w)  ; put at low end
           tst.l  d0             ; any more digits?
```

Instruction **dbeq** is a wonderful three-way operation. First, it decrements (subtracts one from) register D2. Secondly, it won't loop back if an 'equals' condition exists (register D0 contains zero). Thirdly, it will refuse to loop if D2 has gone below zero.

That's a great combination. We needed to decrease the contents of D2. If register D0 contains a zero, we don't need to calculate any more digits: zero suppression happens automatically! And if D2 falls below zero (it shouldn't, in this case) we have no more space in our buffer and wouldn't want to loop again.

```
           dbeq   d2,DigLoop        ; yes, go back
           rts

MsgNewLine   dc.b   $0a
NewLineLen   equ    *-MsgNewLine
SpaceMsg     dc.b   $20
SpaceLen     equ    *-SpaceMsg
InfoMsg      dc.b   '.info.'
DirMsg       dc.b   '       DIR  '
DirLen       equ    *-DirMsg
PromptMsg
     dc.b 'FORMAT:  SeeDir <Directory-name>',$0a
PromptLen    equ    *-PromptMsg
EmptyMsg     dc.b   '....Empty....',$0a
EmptyLen     equ    *-EmptyMsg
CountMsg     dc.b   'items.',$0a
CountLen     equ    *-CountMsg
NoMemMsg     dc.b   'Out of Memory',$0a
NoMemLen     equ    *-NoMemMsg
NoFileMsg    dc.b   'Can t find file.',$0a
NoFileLen    equ    *-NoFileMsg
CtrlCMsg     dc.b   '^C',$0a
CtrlCLen     equ    *-CtrlCMsg
dosname      dc.b   'dos.library',0
ConName      dc.b   'CON:10/0/480/100/SeeFile',0
        end
```

HOURS:
8 AM to 7 PM Monday - Friday
8 AM to 4 PM Saturday
Call on Sunday

The Software Shop, Inc.

Your AMIGA Software and Hardware Marketplace

Visa and MasterCard Accepted

Hardware Spectacular! Sensational Savings! Compare!

HardCards w/HF2000		HardCards w/FastCard		FastTrack A500/A1000		ToolBox A500/A1000		Memory Expansion A2000/A2500/A500/A1000			
HardFrame	$ 249	FastCard	$ 159	FastTrack Sys.	$ 379	ToolBox	$ 249	8Up 2Mg	$ 489	Starboard Ok	$ 269
138N 32Mg	589	138N 32Mg	506	138N 32Mg	722	ToolBox w/48Mg	849	8Up 4Mg	769	Starboard 512K	359
157N 48Mg	669	157N 48Mg	589	157N 48Mg	799	ToolBox w/32Mg	769	8Up 6Mg	1,059	Starboard 1Mg	449
Quantum 40Mg	759	Quantum 40Mg	689	Quantum 40Mg	899	ToolBox w/Q40	939	8Up 8Mg	1,349	Starboard 2Mg	669
Quantum 80Mg	1,089	Quantum 80Mg	1,039	Quantum 80Mg	1,223	ToolBox w/Q80	1,279	Spirit 512K	319	MiniMeg 512K	378
Quantum 105Mg	1,179	Quantum 105Mg	1,099	Quantum 105Mg	1,304	ToolBox w/Q105	1,349	Spirit 1Mg	411	MiniMeg 1Mg	499
277N 63Mg	759	277N 63Mg	689	277N 63Mg	889	ToolBox w/63Mg	939	Spirit 105Mg	499	MiniMeg 2Mg	735

External 5.25 Drv. $179 External 3.5 Drv. $119 Internal 3.5 Drv. $89 Digi-View Gold, WV1410 Panasonic camera w/lens and Copy stand $399

ACCESSORIES

A-B Switch cable	$35.00
A/B/C/D Switchbox	63.00
Boing optical mouse	109.00
Copy Stand	65.00
Ece Midi 1000	52.00
Ece Midi 500/2000	52.00
Epyx 500JX	15.00
Ergo Joystick	17.00
Freedom stick	65.00
Infared mouse	176.00
MD120 3.5 Disk holder	29.00
MD60 disk holder	26.00
MD64 disk holder	26.00
Modem cable A1000	15.00
Modem cable A2000/500	15.00
Mouse holder	6.00
Mouse master	29.00
Mouse pad	9.00
Multisync cable I-102M/F	24.00
Okimate black ribbon	5.00
Okimate color ribbon	8.00
Power strip 6 outlets	17.00
Printer cable 2000/500	15.00
Printer cable A1000	15.00
Quick shot Joystic	15.00
Ribbon panasonic all	10.00
Slick stick	8.00
Star NX black ribbon	8.00
Star NX color ribbon	12.00
Tilt and swivel stand	21.00
Xerox 4020 500/sheet	25.00

CAD SYSTEM

Draw 2000	$183.00
Home builder cad	129.00
Home builder choice	53.00
Intro cad	51.00
Logic works	65.00
Pro-Board	399.00
Pro-Net	399.00
X-Cad system	385.00

A500/A2000/A2500 PERIPHERALS

A1084S stereo monitor	CALL
A2000 HD 40mg28ms w/cont	CALL
A2058 2mg 8mg ram board	CALL
A20620/2meg 32bit	CALL
A20620/4mg 32bit Mem	CALL
A2090A HD controller	CALL
A2300 Genlock	CALL
A2500 HD40mg28ms3mgMem 68020/68881	CALL
Amiga 2000 Cpu	CALL
Amiga 500 CPU	CALL
AT Bridge card A2286	CALL
Bridge card A2000	CALL

COMMUNICATIONS

Atalk III	$65.00
Atredes profess.BBS	96.00
BBS pc	96.00
Diga	50.00
Online Platinum	69.00

CREATIVE PRODUCTIVITY

Analytic art	$39.00
Gizmoz enhanced	45.00
Intellitype	35.00
Magellan	129.00
Master type	25.00
Mavis beacon typing	33.00
Your family tree	32.00

DATABASE MANAGEMENT

Acquisition	$189.00
Data retrieve	51.00
Data Retrieve Prof.	189.00
Dbman	125.00
Micro base	24.00
Micro fiche filer	64.00
Microfiche filer plus	115.00
Organize	49.00
Softwood file II gs	79.00
Super base	52.00
Superbase pers. II	98.00
Superbase Pro. 3.0	229.00
Superbase professional	189.00

PROGRAMMING LANGUAGES

A/C basic	$129.00
A/C fortran	195.00
Algebra ii	33.00
Arexx	33.00
Assem pro	65.00
Aztec C developer	195.00
Aztec C professional	129.00
Aztic complete devel.sys.	315.00
Benchmark C library	60.00
Benchmark iff library	62.00
Benchmark modula 2	128.00
Benchmark simplified	62.00
Cape 68k	59.00
Critic's Choice	159.00
Devpac Ed/Assem/link/debg.	66.00
Dissasembler	45.00
GFA basic 3.0	95.00
Inovatools #1	54.00
Lattice C	132.00
Lattice C + +	379.00
Lattice C professional	249.00
Lattice developemnt system 5.0	199.00
Library's source	197.00
Macro assembler	65.00
Metacomco lisp	131.00
Metacomco pascal	95.00
Metacomco shell	40.00
Metascope debugger	65.00
Power windows v2.5	58.00
Pro Basic	124.00
Publishers choice	130.00
Source level debugger	50.00
The power system pecan	93.00
The professional pecan	89.00
The Works Platinum	164.00
True basic	65.00
W. shell	33.00

FONTS

Asha's fonts	57.00
Calligrapher	$63.00
Fancy 3d fonts	52.00
Font set I	22.00
Inter font	76.00
Kara fonts color	50.00
Lion fonts	56.00
Masterpiece fonts	159.00
News letter fonts	29.00
Profonts I Prof	23.00
Profonts II decorative	23.00
Studio font	29.00

HARDWARES

8Up Memory Exp. A2000	$165.00
Accelerator	152.00
Amigen	153.00
Easyl A1000	349.00
Easyl A2000	379.00
Easyl A500	345.00
Expan. 512K A500	159.00
Expansion 1mg A500	409.00
Flicker fixer	495.00
Framegrabber	579.00
Future sound	129.00
Light pen	99.00
Live A2000	349.00
Live framegrabber A1000	245.00
Live framegrabber A500	305.00
MC68881 Math Co-processor	179.00
Multifunction card	75.00
My-T-Mouse	39.00
Perfect sound A1000	69.00
Perfect sound A500/A2000	69.00
Phoenix power supply A500	95.00
Progen	379.00
Super Gen	695.00
Supra modem 2400bd	149.00
WV1410 Panasonic Camera w/lens	199.00

SPREADSHEETS

Analyze! 2.0	$95.00
Haicalc	32.00
Maxiplan 500	95.00
Maxiplan plus	126.00
Super plan	97.00
VIP Professional	65.00

DESKTOP PUBLISHING

Citydesk	$97.00
Citydesk companion	15.00
Citydesk companion II	15.00
Page stream	129.00
Page setter	97.00
Professional Draw	129.00
Professional page	249.00
Publisher plus	69.00
Shakespeare	129.00

ACCOUNTING

The Accountant	$183.00
Best general ledger	77.00
Financial plus	189.00
Invester advantage	63.00
Miamiga ledger	63.00
Micro lawyer	39.00
Easy Ledger	199.00

WORDPROCESSORS

Becker text	$92.00
Excellence!	179.00
Flow idea processors	63.00
Gold spell II	30.00
Kindwords	65.00
LPD writer	75.00
LPD writer unprotected	83.00
Micro text	22.00
Prowrite v2.0	80.00
Reason	249.00
Text pro	50.00
Word perfect	159.00
Wp library	79.00

MONITORS

Amdek multisync 12' mon.	$512.00
Panasonic 8cm515 mon.	249.00
Sieko cm1430 monitor	699.00
Sony multi-scan Monitor	535.00
Zenith 14' Flat screen mon.	720.00
Nec 3D multisync	799.00
Taxan 1000 20" multisync	2899.00

EDUCATION

Adventure of sinbad	32.00
Aesop's fables	32.00
All about America	35.00
Animal kingdom	32.00
Decimal dungeon	32.00
Discovery game math	25.00
Discovery game spell	25.00
First letters and words	35.00
First shapes	35.00
Fraction action	32.00
Grade manager	59.00
Great states	25.00
Kid Talk	35.00
Kinderama	32.00
Kwik speak spanish	19.00
Learning curve	52.00
Link word French	22.00
Link word German	22.00
Link word Italian	22.00
Link word Russian	22.00
Link word Spanish	22.00
Math talk	35.00
Math talk fraction	35.00
Math wizard	35.00
Perfect score	51.00
Quiz master	51.00
Read & rhyme	32.00
Read-a-rama	32.00
Spell bound	25.00
Speller bee	35.00
Tales from Arabia	32.00
Wordmaster	32.00

PRINTERS

1091i Panasonic printer	$224.00
Alps Allegro 24 PIN	399.00
Epson Apex 80	205.00
NEC LC890 laser PS.	3395.00
Okimate 20 color prt.	145.00
Plug'n print	85.00
Star nx rainbow	259.00
Xerox 4020 ink jet	1295.00
Xerox 4020 starter kit	159.00
HP Paint jet	CALL

1-800-752-0050

In Mass. Call....

1-508-756-6452

SALES AND QUOTES (outside Mass.):
1-800-752-0050

SHIPPING AND TECHNICAL SERVICE:
1-508-756-6452

QUOTE-A-FAX: 1-508-799-9354

- Corporate and volume purchases invited. Call Moe.
- Stock orders placed by 12:00 NOON EST shipped the same day.
- Allow 10 days for personal checks to clear.
- Wire transfer accepted.
- Minimum $5.00 Shipping/handling charge
- Add $3.50 for COD.
- Returns subject to restocking charge
- Software defects-call the Manufacturer
- For faster delivery send cashiers check

SOUND & MUSICS

4-op deluxe	$97.00
Audio master	37.00
Audio master II	67.00
Back songbook	27.00
Copyist II	179.00
D-50	99.00
Deluxe music	69.00
Dr drums	20.00
Dr keys	20.00
Dr T's Keyboard	160.00
Dr. T's Midi recording studio	45.00
Dx heaver	97.00
Dynamic drums	52.00
Dynamic studio	129.00
Future sound A500/A2000	92.00
Hot cool jazz	18.00
Hot licks	32.00
Instant music	32.00
Matrix 6	97.00
Midi Magic	99.00
Mt-32	97.00
Music student	38.00
Pro sound designer	125.00
Promidi studio	129.00
Rock & roll	21.00
Sonix	51.00
Sound sampler	78.00
Studio magic	65.00
Synthia	59.00
Texture	97.00
Utilities 2(mimatics)	43.00

UTILITIES

B.A.D.disk optimizer	$32.00
C.B. Tree	65.00
Celebrity cook book	20.00
Custom screens *	45.00
Cygnused Professional	65.00
Disk 2 disk	33.00
Disk master	33.00
Disk mechanic	59.00
Dos 1.3	22.00
Dos 2 dos	35.00
Doug's math aquarium	52.00
Encore	45.00
Ez backup	33.00
Facc II	22.00
Fine print	33.00
Fleet check	33.00
Flip side	37.00
Gold spell	32.00
Gomf	27.00
Grabbit	19.00
Laser scripts	28.00
Laser up: fonts	26.00
Laser up: plot	33.00
Laser up: print	58.00
Laser up: utilities	26.00
Lex check	33.00
Lv backup	32.00
Multi preferences	20.00
Nag Plus Schedule Assist.	52.00
Pro script	32.00
Project D	32.00
Promis spelling	32.00
Quarterback	45.00
Raw copy 1.3	39.00
Superback	52.00
Sys font *	22.00
Tax plan	22.00
Tdi editor sources	30.00
Tdi examples	15.00
Tdi grid access	30.00
Tdi kermit	19.00
Text ed plus	52.00
V.I.P.	32.00

HOME ACCOUNTING

Desktop budget	$46.00
Money mentor	$65.00
Phaser	65.00
Tax Break	53.00

GRAPHICS & VIDEO

3-Demon	$69.00
Animate 3-D	99.00
Animate 4D	439.00
Animation editor	39.00
Animation effects	32.00
Animation Flipper	32.00
Animation multiplane	58.00
Animation stand	32.00
Animation w/images	89.00
Animator apprentices	185.00
Animotion	65.00
Architechtural design	23.00
Art gallery I	20.00
Art gallery II	20.00
Art pak I	20.00
Art parts 1	20.00
Art parts II	20.00
Award maker plus	32.00
Broadcast Titler	189.00
Butcher	25.00
C light 3d editor	39.00
Calligrapher help	28.00
Chroma Paint	48.00
Color splitters	79.00
Comicsetter	65.00
Deluxe paint II	89.00
Deluxe Paint III	102.00
Deluxe print II	55.00
Deluxe photo lab	102.00
Deluxe production	137.00
Deluxe video II	89.00
Designasaurus	32.00
Digi paint	39.00
Digi paint help	24.00
Digi view	131.00
Director's Toolkit	26.00
Dpaint help	24.00
Exprss paint	65.00
Express paint III	89.00
Forms in flight II	77.00
Future design 3-d	23.00
Graphic studio *	35.00
Human design 3-d	23.00
Icon Paint	59.00
Impact	55.00
Interchange	32.00
Invision	109.00
Lights Camera Action	49.00
Microbot design 3-d	23.00
Modeler 3d	64.00
Movie setter	65.00
Page flipper fx	95.00
Perfect vision	175.00
Photon expansion disk	19.00
Photon paint	65.00
Photon paint 2.0	97.00
Photon paint help	23.00
Photon video cell anim.	97.00
Pixmate	43.00
Print master plus	33.00
Prism	43.00
Pro video plus set I	83.00
Pro video plus set II	83.00
Pro. result using dpaint	35.00
Provideo plus	193.00
Sculpt 3-d	64.00
Silver turbo	129.00
The Director	46.00
Turbo silver	129.00
Tv-show	64.00
Tv-text	64.00
Video effects 3d	121.00
Video scape 3-d	120.00
Video titler	95.00
Zoetrope	91.00

DISKETTES

Fuji 3.5' ds/dd color disks	$19.00
Sony 3.5' ds/dd	19.00
TDK 3.5' ds/dd disks	19.00
Centech 3.5' ds/dd disks	19.00
Centech 5.25' ds/hd disks	29.00

ENTERTAINMENT SOFTWARE

We have them all! Over 300 games in stock. Call for your favorite game, and check our low prices.

SHOWROOM: 22 Front Street, Worcester, MA 01614

A File System Case Study

The Software Distillery's network file system

by Doug Walker
Copyright © 1989 Doug Walker.

Doug Walker is a Senior Systems Developer at SAS Institute, Inc. in Cary, North Carolina. He has been programming the Amiga since 1986 as a member of The Software Distillery, which is responsible for many popular freely redistributable programs including Hack and Hack Lite, PopCLI, MemWatch and Blink. He can be reached on PeopleLink (DWALKER) and Usenet (mcnc!rti!sas!walker).

In the last two issues, John and I have led you through file systems from the program's point of view and from the system's point of view. In this article, I will describe some problems we encountered when implementing a network file system, and how we coped with them.

The file system described here is still under development. A first version has been released and is freely redistributable, but it supports only two nodes in the network. It also requires Matt Dillon's DNET package to run. When the NET: handler is complete, it will work with any communication method (including DNET) for which a network communications driver is written. Several hardware vendors have already promised to support the effort. (If you are developing Amiga network hardware, please contact Doug Walker or John Toebes for more information.) Since NET: is independent of the communications method used, I will simply assume in this article that some method of getting data from machine A to machine B is available.

The big picture

Before describing the handler itself, I should explain how we want the network to appear to the system. A new device called NET: would be available. If you double-clicked on its icon or did a **dir** of NET:, you would see a directory for each other computer (or 'node') in the network. Going one level deeper, each node's directory would contain subdirectories for each device on the remote node. Thus, if I call my machine DWALKER, and John calls his machine JTOEBES, I can get a directory of his hard drive by typing:

```
dir net:jtoebes/dh0
```

Since this gets fairly cumbersome, I would probably want to say:

```
assign john: net:jtoebes/dh0
```

so I could access his hard drive with:

```
dir john:
```

Since the NET: handler will support multiple communications drivers, some of the nodes accessible via the NET: device may be available via the serial port, some via an AppleTalk or DECNET board, and so forth. NET: will pull them all together.

Internal data structures

As a file system, NET: is unique in many ways. It has no fixed media beneath it, as does a floppy drive or a hard drive. Part of its data may be accessible while another part is not, if one node crashes while another stays up. It cannot use read buffers, FileHandles, FileInfoBlocks and so forth supplied by the application because the remote side cannot get to them. To support this diversity, we had to come up with some data structures that allowed us to distinguish between network nodes and to determine where to send data when it is routed to a node. To do these jobs, the handler keeps a linked list of NetNode structures describing the nodes. (See the box at the top of the next page for the definition of the NetNode structure.)

The *status* field indicates the last known status for the node. Possible values are #defined after the structure definition.

The *name* field contains the name of the remote node, stored as a null-terminated BCPL string. This way of storing the node name gives us a valid BCPL string to use when we need it in that form, and also a normal C null-terminated string, when we need one, simply by using *name+1*.

The *id* field contains some kind of identifier unique to the node that is guaranteed to change if the node reboots or shuts down. This field is not currently used, but it will be used in the future

to verify that a lock or filehandle allocated by a remote system still exists before using it. If you allocate a lock on the remote node, then the remote node crashes and reboots, your lock will be invalid; the *id* field allows us to detect this condition.

In future releases, the *ioptr* field will be used by the communications driver to get any driver-specific information it needs about the node; this allows the same network handler to access nodes via many different routes. This means that an Amiga with an AppleTalk board in it could use the network handler to access nodes via AppleTalk, and use the same handler to access nodes via the serial port.

Notice the definition of the struct *NetPtr* as a substructure. Struct *NetPtr* is our solution to one of the basic problems any file system faces: how you will uniquely identify files using only the four-byte *fl_Key* value in a *FileLock* structure. File systems handling floppies or hard disks usually use the sequential block number of the first block of the file. In a network file system, that obviously will not work. We chose to use the *fl_Key* value as a pointer to a *NetPtr* structure instead. Each lock created and tied to NET: has a struct *NetPtr* allocted for it that points to the *NetNode* structure for that lock or filehandle. The instance of struct *NetPtr* in the *NetNode* structure itself is useful for handling NULL locks. In addition to the pointer to *NetNode*, the *NetPtr* structure contains two RPTRs, which are pointers to memory on the remote node. The first points to the remote device that is responsible for the lock; the second points to the actual lock on the remote node. The information in the *NetPtr* structure is passed to a server program on the remote node, which then sends packets to the specified device to perform the requested actions. (The functions of the server will be discussed in more detail later).

In addition to locating the file based on the *fl_Key* field of a struct *FileLock*, we must be able to locate the file based on the *fh_Arg1* field of a struct *FileHandle* in order to do I/O. Disk-based handlers need to keep a lot of status information on the file; this is not a problem for the network handler, since another handler on the remote node is keeping that for us. All we need is a pointer to the *FileHandle* on the remote node. The *NetPtr* structure is designed to do exactly this. When a file is opened, a struct *NetPtr* is allocated and the *fh_Arg1* field of the *FileHandle* is set to point to the created structure. The *RPtr* field of the *NetPtr* structure is set to the *FileHandle* on the remote node.

Example: locking a file

Having defined some of the data structures, it's time to discuss some specifics. I will take the ACTION_LOCATE_OBJECT, or *Lock*, packet as an example, and trace it from the time it arrives at the handler to the time it is returned to the caller.

Let's say that we have the network set up as described previously and we issue the command:

```
cd net:jtoebes/dh0/work
```

```
#define RNAMELEN 16            /* Maximum nodename length              */

typedef char *RPTR;            /* Pointer to memory on another node    */

struct NetNode
{
   struct NetNode *next;       /* Next Net Node in chain               */
   int status;                 /* Last known status - see defines      */
   char name[RNAMELEN+2];      /* Name of remote node                  */
   short id;                   /* ID of remote node (timestamp)        */
   APTR ioptr;                 /* Driver-defined pointer               */
   struct NetPtr
   {
      struct NetNode *NetNode; /* Ptr to network address information   */
      RPTR RDevice;            /* Address of remote MsgPort            */
      RPTR RPtr;               /* Remote file system's lock/filehandle */
   } RootLock;
};

/* defines for NetNode status field */
#define NODE_DEAD    0     /* The node has never been heard from       */
#define NODE_UP      1     /* The node is up and responding normally   */
#define NODE_CRASHED 2     /* The node was responding, but is now down */
```

The C definition of the *NetNode* structure.

At some level, an ACTION_LOCATE_OBJECT packet will be sent to NET: by the **cd** command to lock the directory you are changing to. This packet contains three parameters, as decribed in the first article of this series: a lock that this lock is relative to, the name of the file or directory to lock, and an access type (ACCESS_SHARED or ACCESS_EXCLUSIVE). In this example, the relative lock will be NULL, indicating the lock is relative to the root of the device, and the name will be "net:jtoebes/dh0/work". The type will be ACCESS_SHARED. The sequence of events in this case would be:

1. The handler determines which network node is to receive the packet. It gets the name of the node by looking at the specified filename, then searches the linked list of *NetNode* structures for a node with the right name.

2. Once it finds the correct node, it strips everything up to the node name out of the provided filename. The lock request is passed on to the remote node, with the stripped name ("dh0/work") passed as the name to lock.

3. The remote node's server gets the lock request. Since there is no relative lock, it must determine what device on the remote node this request is for. It strips off everything before the '/' to get the device name (in this case, "dh0").

4. The server calls the *DeviceProc* to get the *MsgPort* for DH0. It sends an ACTION_LOCATE_OBJECT packet to DH0 with a null relative lock and what is left of the filename ("work"). The file system for DH0 allocates a lock and returns a pointer to it.

5. The server notifies the handler on the local node of the results of the transaction. It also returns the pointer to the remote lock and the address of the *MsgPort* on the remote node.

6. The handler allocates a *NetPtr* structure. It points the *NetNode* field to the *NetNode* structure associated with the remote node. It sets the *RDevice* and *RPtr* fields to hold the address of the remote *MsgPort* and the remote *Lock*, respectively. It also allocates a struct *FileLock*, and sets the *fl_Key* field to point to the *NetPtr* structure.

7. The handler returns the *FileLock* structure to the caller.

Notice some interesting things about this sequence. The caller doesn't know that the *FileLock* structure locks a file on the remote node; as far as he's concerned, it's just like the *FileLock* structures returned by any other handler. The DH0 file system on the remote node doesn't know that the just-created lock was created on behalf of a process on a different machine; as far at it's concerned, the lock was on behalf of the server process.

Example: examining a file

Now let's say we issue the **dir** command after doing the **cd** listed above. The CLI will send an ACTION_EXAMINE_OBJECT packet. The two parameters to this packet are a lock on the object to be examined and a struct *FileInfoBlock* to be filled in. The sequence in this case would be as follows:

1. The handler determines which network node is to receive that packet by looking at the *fl_Key* field of the *FileLock* structure. It points to the *NetPtr* structure allocated before. The field *NetPtr->NetNode* tells the handler which network node to send the information to.

2. The handler sends the ACTION_EXAMINE_OBJECT request to the remote node's server. It also sends the *RPtr* and *RDevice* fields from the *NetPtr* structure.

3. The server gets the examine request. It allocates a *FileInfoBlock* and sends an ACTION_EXAMINE_OBJECT packet to the specified *RDevice*, using the specified *RPtr* as the lock.

4. The examine packet is returned to the server. The server sends the data in the *FileInfoBlock* back to the handler on the local node.

5. The handler copies the data in the *FileInfoBlock* into the caller's *FileInfoBlock* and returns the packet to the caller.

Example: reading a file

As one final example, let's open a file, read some data, and close it. This would be done if we issued the command **type readme** after having done the **cd** command above.

The first packet is an ACTION_FINDINPUT. It takes a pointer to a *FileHandle* structure, a *FileLock* and a name. The *FileHandle* structure is allocated by the caller; the *Lock* is the lock we created before. The name will be "readme". The packet is processed as follows:

1. The node to receive the packet is determined from the lock, as above. The ACTION_FINDINPUT request is sent to the remote server.

2. The remote server allocates a struct *FileHandle* on the remote node and sends an ACTION_FINDINPUT request to the correct file system. It sends the results back to the local handler.

3. The local handler allocates a *NetPtr* structure for the new *FileHandle* and sets the *fh_Arg1* field to point to it. No other fields are changed in the *FileHandle* structure. The *NetPtr* structure is initialized with information returned from the remote node.

4. The packet is returned to the caller.

Once the file is open, we issue an ACTION_READ packet. Let's say the read request is for 100K - not an unreasonable size for a read block under the 1.3 Fast File System.

The ACTION_READ packet takes three arguments: the *fh_Arg1* of the open *FileHandle*, the address of the data buffer to use, and the number of bytes to read.

1. The node to receive the packet is determined from the *fh_Arg1* passed in.

2. Now we should pass the read request to the remote node. However, if we do so, we will force the remote node to allocate 100K of memory to read the data into. This is not desirable, since the remote node may be tight on memory. Therefore, we break the 100K read request into smaller pieces. Eventually, a node will set its maximum read amount when it is mounted; for now, the maximum read amount is always 1K. Since the time spent actually reading the device is dwarfed by the time spent transmitting the data, this is not too unreasonable.

3. The remote node gets a 1K read request and passes it on to the appropriate remote device. It passes the results back to the local node.

4. The local handler copies the 1K read into the provided 100K data buffer, then goes back to step 3 until the buffer is full or end of file is reached. When done, it returns the packet to the caller.

Assuming that the file has less than 100K, we should be done reading. The next packet is therefore an ACTION_END, which closes the open *FileHandle*. The only argument is the *fh_Arg1* of the open *FileHandle*.

1. The local handler sends the ACTION_END request to the remote server.

2. The remote server sends it to the remote device.

3. The remote server frees the struct *FileHandle* it had allocated to hold the *FileHandle* on the remote node. It then returns the results to the local node.

4. The local handler frees the *NetPtr* structure it had allocated and nulls the *fh_Arg1* field. It then returns the packet to the caller.

Odds and ends

The other packets are handled similarly. The handler determines which remote node to send to and passes the request on; the server constructs a DOS packet with the request and sends it to the correct device; the results are returned to the handler, and in turn to the user.

One slight complication comes in when the user refers to files in NET: itself. For example, if the user types **dir net:**, the **dir** command will expect to be able to *Examine* and *ExNext* objects in NET:. And what happens if the user types **copy myfile to net:**?

Our network handles the second problem by not permitting write access in the root of the network. The first problem is thereby made much simpler as well.

If you do an *Examine* on NET:, it fills in information as requested. If you continue to *ExNext* NET:, you will get information on each node in turn. **.info** files are also reported for each network node. This is necessary in order to make the network function with Workbench, since the **.info** file for an object must reside in the same directory.

If one of the **.info** files is opened, locked, or read, NET: opens the file "Node.info" in the root of the remote node instead. This allows the owner of each node to select a customizable icon for his or her Amiga, and have that icon appear on all other users' Workbench screens whenever that Amiga is accessed.

Another complication is introduced by the ACTION_INFO and ACTION_DISKINFO packets. NET: reports its size as one block per node in the network, and its percent free as the number of nodes in the network that are currently up. This allows you to tell at a glance how many network nodes are active simply by looking at the empty/full disk indicator on the NET: window on the workbench screen, or by looking at the amount of 'free' space in the output from the **info** command.

There are many small points like the last two to cover but, as in most programming projects, the only complete documentation is the source code. The source code to this first version is available on this month's disk, in Lattice C. Hopefully, the complete NET: will be available by the fall, including multiple nodes, multiple communications servers and improved performance. As always with Software Distillery projects, NET: will be freely redistibutable, so enjoy!

Others Make Claims...ASDG Makes Deliveries!

ASDG's Dual Serial Board

Use *ANY* Program Written For "serial.device" On *ANY* Expansion Port.

● Two high speed RS-232C serial ports for the Amiga 2000. ● IBM PC AT style (9 pin male) connector for each port. ● Full 7 wire implementation plus Ring Indicator and DCD. ● Send and Receive 110 to 76800 baud (and higher). ● Complete Amiga system software (both Exec *and* DOS). ● Using our exclusive Serial DisPatcher, programs can use any port *without* modification. ● Public domain and shareware software on the distribution diskette.

ASK YOUR DEALER!

ASDG

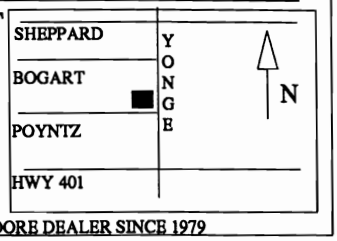
925 Stewart Street
Madison, Wi. 53713
(608) 273 - 6585

Amiga is a trademark of Commodore-Amiga, Inc. IBM PC AT is a trademark of IBM Corp.

CANADA'S BEST SELECTION OF AMIGA - C64 - C128 SOFTWARE
COMSPEC HARD DRIVES AND BERNOULLI DRIVES AVAILABLE FOR THE AMIGA 500 AND 2000

Electronics 2001
COMPUTER PRODUCT CENTRE

4736 YONGE STREET
WILLOWDALE ,ONT
M2N 5M6
TEL: (416) 229-2700



AUTHORIZED COMMODORE DEALER SINCE 1979

New Store!

Fisher's
Computers & Software

Authorized Amiga Dealer

8005 Archibald
Rancho Cucamonga, CA 91730
(714) 987-1662

Serving The Inland Empire

The FRANCES Memory Board

Adding 32-bit RAM to the LUCAS board

By Brad Fowles and Eric Haberfellner

Back in Volume 1, issue 3 *Transactor for the Amiga*, I wrote an article describing a 68020/68881 accelerator board of asynchronous design called LUCAS (Little Ugly Cheap Accelerator System). Now I'd like to introduce you to FRANCES (Fast Ram At Nominal Cost, for Extended Storage).

FRANCES is a 32-bit wide memory board that fits onto the expansion connector of the LUCAS board. You should read the original LUCAS article for a more complete understanding of the project as a whole.

The raw computing power of the LUCAS/FRANCES combination is pretty impressive. It will turn your Amiga 1000 into a full 32-bit platform running at clock speeds from 16 to 20 MHz (MegaHertz). Like the LUCAS board, I will make available a bare board, 2 PALs, hardcopy schematic and documentation disk for $75.00 US. (See the end of the article for more information.) If you've got the urge to hack or, like me, just can't afford a full blown A2500, this project will tweek a wee bit more life out of the A1000.

I should state my bias right up front. My real interest in the Amiga is as an experimental platform for 3D character animation. My needs, therefore, are for raw computing power and the project shows this bias.

Before we get into details, let me thank all of you who participated in an experiment in public domain hardware. To date I have sent out 550 LUCAS boards. Some of you had problems with the project, and a few had no luck at all; but judging by the mail I receive, most of you seemed to get the project up and running, and were able to improve the performance of the A1000. Now we are going to see if we can make the grand ol' lady fly.

Eric Haberfellner, who is co-authoring this article, not only wrote the *AFM* memory utility and all the diagnostic software to help me design the system, he stayed up nights helping to get it to work . I thank him for all his past work and all the answers he'll be giving to you folks when the time comes. My only complaint is that he and his brother slowed down the development of the first running prototype by booting up with the *FA/18* flight simulator whenever my back was turned. By the way, at 20 MHz the FA/18 demo runs in a little under 3 minutes, compared to 5 minutes, 47 seconds on a stock Amiga.

The thing that really kept the project going was the support you all gave to each other. I was constantly heartened by calls from local user group hardware gurus who were helping their friends get their systems up. LUCAS is by no means perfect. There were problems with some peripherals, bus noise problems and the now famous selection of U9. But with true hacker spirit, most of you found a way to sort out the problems, and hopefully learned a little along the way.

In the original article I said that the main benefit of having an '020 in your machine was that it provided a 32-bit wide upgrade path. However, until FRANCES entered the picture, the only 32-bit data transfers were those that were happening between the 68881 math coprocessor and the '020.

This was great for floating point math but really did little for general purpose computing. All memory transfers were still only 16 bits wide. What we really needed was a 32-bit wide memory system to support the '020.

Design criteria

The first decision to be made was: "How much memory do we need?" (All we can get, right?) I wanted to keep the board to a form factor that would be able to sit in the A1000 and still allow the case to be put on. I didn't want to use those wonderful little SIMMs (Single Inline Memory Modules) because they can be hard to find, and can be quite expensive in small quantities. I looked at the space available and decided that I could put in 4 Mb (megabytes) of standard 20-pin DIP (dual inline package) memory chips without too many problems.

Next: "How fast?" I priced out static memory fast enough to operate with no wait states – back in February, '89 the price would have been $1200.00 per megabyte.

So, OK, we'll forget about that, what about DRAMs (dynamic random access memory)? At $25.00 per chip, that's $200.00 per megabyte (at the time of writing, prices have dropped to $19.00 U.S. per chip). That seemed a little more reasonable.

One-megabit by 1-bit chips are slightly cheaper than 256-Kbit by 4-bit chips, but with the former you would need to buy 32 chips right off. If I used 256-Kbit by 4-bit chips you could start out with eight chips (for one megabyte) and build up in one megabyte increments. Yeah! Better to do it that way.

Keeping price in mind, how fast should the memory be? It turns out that 100-ns. (nanosecond) DRAMs will operate at 16 MHz with only one wait state and at 20 MHz with two. Even if we were to use 70-ns. DRAMs, the same number of wait states would be required. Keeping these trade-offs in mind, I decided that 256-Kbit by 4-bit, 100-ns DRAMs were the best choice; but check the price of 70-ns. parts – the difference may be negligible.

How does the change from 16-MHz to 20-MHz clock speed affect performance? The '020 has a three-cycle access. At 16-MHz, three times 62.5 ns., plus one wait state of 62.5 ns., equals 250 ns. total. At 20 MHz, it's three times 50 ns., plus two wait states of 50 ns., which equals, let's see, uh... 250 ns. Hmmmmmmm! At either 16 MHz or 20 MHz, the effective memory speed will be equal. Of course, at 20 MHz the transfers between the '881 and the '020 will be 1.25 times faster, as will all the internal register stuff in the '020 itself. It is possible that running at 18 MHz with one wait state, using 70-ns. DRAMs, would yield the best performance. In a simple test run at presstime, this does appear to be the case.

Now we need to select a DRAM controller. I remember, just a few short years ago, designing DRAM control circuitry with a plethora of discrete components and delay lines. These days there is a wide variety of highly integrated controllers from which to choose. I decided on a controller which could be soft-programmed to accommodate various clock speeds and DRAM configurations, and chips made by different manufacturers. I also wanted a controller which could support memory interleaving – more on that later. The best choice, I believe, is the National Semiconductor 8421A-20 or -25. So, with the controller and memory chosen, and a bit of glue and a few buffers added, we're almost there.

In order to squeeze all the performance we can out of the memory board, it would be desirable to get all those routines that are in the KickStart area of memory out of 16-bit wide space and into 32-bit space. Since we don't have an MMU to do the address translations for us, we're going to have to do it some other way. To describe how this is done, it's time to go a little deeper into the technical details. If you know nothing of hardware, hang in there and I'll try to give enough background so that it should make sense.

Practising 'Not Being Seen'

I assume that you have read the article on the LUCAS board. When the '020 is accessing the '881, it does so at full speed and the data path is 32-bits wide. If the circuitry of the Amiga sees this access, it will become quite confused and respond with a *DTACK (Data Transfer Acknowledge) long after the original cycle has gone by. To prevent this, we simply don't tell Amy that this is going on. This is accomplished by not asserting *AS (Address Strobe) or either of the two data strobes, *UDS or *LDS (Upper Data Strobe, Lower Data Strobe). This is done in PAL U7 on the LUCAS board by conditioning the term for these signals with the *CPCS signal (Coprocessor Chip Select) nonasserted state. In this way, the '881 practises Not Being Seen by the Amiga system underneath it.

The FRANCES memory board has the same type of restrictions: because it is much faster than the normal Amiga memory, we don't want Amy to try and respond to these faster accesses, so, same answer, we don't let her know about them.

In order to effect these clandestine accesses, we take the *CS (Chip Select) signal from the FRANCES decode PAL and run it back to U7. It just so happens that the U7 PAL on the LUCAS board receives two signals from the LUCAS expansion connector: *SRDSACK0 and *SRDSACK1 (Static RAM Data Size Acknowledge). I had put these lines in before I realized the costs of high speed static RAM. So I redefined them. The *SRDSACK0 line became the *FRANCYC (FRANCES Cycle) line and the *SRDSACK1 line became *FDSACK (FRANCES DSACK). Only one *DSACK line is needed because the FRANCES PAL takes care of the byte addressability. We can therefore assume that all memory accesses on the FRANCES board are 32-bits wide. We condition the terms for *AS, *UDS and *LDS with the *FRANCYC line in the same way we did for the coprocessor. Thus, the FRANCES memory practises Not Being Seen and there are no conflicts with the Amiga's decode circuitry.

I started this section by talking about remapping KickStart into 32-bit memory space. To do this, the first thing we must determine is when an access to KickStart memory is about to take place. The '020 initiates a memory cycle by putting the address on the bus and then asserting *AS to tell the system that the address is valid. At 16 MHz the address is valid 15 ns. before *AS is asserted. This is the amount of time we have to do our remapping. (For a little perspective, 15 ns. is the time it takes for light to travel approximately 15 feet.) KickStart resides at the addresses $FC0000 to $FFFFFF (it is also at $F80000 to $FBFFFF). So if we decode an address in which the top five bits are ones, we know that this will be a KickStart access.

If you look at the schematic you'll see that I used a 74F30, eight input NAND gate to decode the address. If the top five bits of the address are ones, the 74F30 will output a low true signal, which I call *REMAPK. If we split this signal into two and invert one of them, one will be low true when there is a KickStart access, and one will be low true when there is a non-KickStart access.

I added two address buffers between the '020 and the DRAM controller on the upper six bits of the address bus. If the '020 puts a KickStart address on the bus, one buffer will substitute the top six bits with the contents of the jumper block that selects the 32-bit KickStart copy address. If the '020 puts a non-KickStart address on the bus, the other buffer passes it through.

The final requirement is that we copy KickStart to the area defined by the jumper block and then tell the circuitry that it is okay to begin remapping. We do this by setting a flip-flop that enables the 74F30 to begin decoding. Again, there is no contention with the Amiga's circuitry because any access to the FRANCES memory will assert the *FRANCYC signal and it will practise Not Being Seen. Nifty, huh?!

Addressing scheme of FRANCES

While designing the board, I did a very non-Amiga thing. The FRANCES memory doesn't autoconfigure. Before you send somebody for a rope, listen to my reasons and the ways I've solved various problems... and then send somebody for the rope.

First: to keep the design as flexible as possible, I used a highly programmable DRAM controller. But autoconfiguring does not allow it to be programmed.

Second: we have this nifty KickStart remapping scheme, but games that take over the system won't be able to use it because they are not able to initialize the KickStart remapping circuitry.

We have found a way that these games can also take advantage of KickStart in 32-bit space. My good buddy Eric will describe how this is done in much greater detail a little later, which is only fair, since he came up with the solutions to all the problems of not autoconfiguring.

While we are waiting for the guy with the rope to get back, let me give you the bad news. There is no provision on the FRANCES board to allow DMA (Direct Memory Access) transfers. There are three reasons for this:

> **Cop-Out #1**: It's a very knotty problem. It necessitates that the memory look like 32-bit wide memory to the '020, but like 16-bit wide memory to the expansion connector. I had a paper design that grew in complexity to the point where I realized that this beast was going to be very difficult to layout on the PCB (Printed Circuit Board).
>
> **Cop-Out #2**: There are very few DMA devices available for the A1000. I couldn't see adding six more months delay to the project for a feature that very few people will use.
>
> **Cop-Out #3**: My dog ate the original paper design... honest, teacher!

FRANCES can be run in or out of the normal Amiga 24-bit address space. Most people have 2 Mb of FAST ram, so if you're running in Amiga space, the board will reside between $400000 and $7FFFFFF. This has the added feature of keeping the decode circuitry as simple as possible.

If you want to run the board outside of Amiga space, perhaps because you have more than 2 Mb of FAST RAM, you can run it from $40400000 to $407FFFFF.

I also used two other high address spaces: $80800000 is the address that controls the flip-flop to enable the KickStart remapping, called ERKS (Enable Ram KickStart); and $80400000 is the address to strobe the programming pin of the DRAM controller. The DRAM controller doesn't have any data lines going to it so it must be programmed using the address bus. This is done by strobing a pin on the controller called *ML (ModeLoad) and placing the value to be put in the 23-bit programming register on the address bus.

The programming details of the DRAM controller are beyond the scope of this article. They are described both on the documentation disk that comes with the FRANCES board and in the 'spec' sheets that are available from National Semiconductor.

Interleaving

There is another way that we can squeeze a little more performance out of the memory system. The memory array is in four banks of one megabyte each. If you have a full four megabytes installed, it is possible to *interleave* the memory accesses to shorten the access time. Under normal circumstances, the top two address bits (in this case, lines A20 and A21) select which bank is active. If, however, we use the bottom two address lines, A2 and A3, then the long word sequential accesses that are the most common type of memory access will happen on separate banks. This allows the DRAM controller to reduce the *RAS (Row Address Strobe) precharge time on the second access. (A0 and A1, and SIZ0 and SIZ1, are not used for memory addressing directly; they are used by the PAL on the FRANCES board to allow byte addressability.)

Interleaving provides a relatively small performance improvement that is highly application specific; but it was easy to do and didn't cost anything. If you only have one, two or three megabytes installed then you cannot interleave the memory. There are jumpers on the FRANCES board to select the interleaving option.

The FRANCES board will take either a right angle or straight 96-pin DIN connector to mate with the LUCAS board. The right angle connector is preferable to the straight one, and is the one I intended. However, I forgot to specify this in the original LUCAS docs, so I made the FRANCES board compatible with both. However, the form factor of the board is slightly different for each of the two connectors. If you are ordering a FRANCES board, make sure you indicate which type of connector you used on the LUCAS board.

The FRANCES board is L-shaped and takes up the rest of the 'second level' inside the A1000. Installing the board requires that you first unplug the power connector that goes from the Amiga's power supply to the motherboard. Once the FRANCES board is in, this power connector plugs into a matching connector on the FRANCES board. Then you take another cable and plug one end into the second connector on the FRANCES board and the other end into the power connector on the motherboard.

AFM: the FRANCES memory configuration utility

Since the FRANCES memory does not autoconfigure, we were faced with the task of making it available to AmigaDOS.

The program usually used to add non-autoconfig RAM is the *Addmem* program written by Commodore-Amiga. There are, however, some problems with this program that make it a less than perfect utility for our purposes:

1. Addmem clears the memory that it adds to AmigaDOS. This makes it unsuitable for use with rebootable ram disks like ASDG's VD0: or AmigaDOS's RAD:, since it clobbers them.

2. Addmem is usually invoked by the s:startup-sequence script file that gets executed as AmigaDOS is booting. By the time it gets executed, some things have been loaded into CHIP RAM that could have been loaded into the FRANCES memory – the trackdisk.device for example. Since CHIP RAM is probably the most precious resource available on an Amiga 1000, and since programs may run slower there than in FAST RAM (depending on system DMA load), we decided that it would be desirable to come up with an alternative approach that would permit us to configure the FRANCES memory as soon as possible in the boot. This would insure that as much executable code as possible would get loaded into the FRANCES memory (and really fly), and that as much CHIP RAM as possible would remain available.

3. Addmem must be run every time that the system is booted or the memory will not be visible to AmigaDOS. This means that any memory made available to the system using the Addmem utility will not be used by applications that require rebooting the system, and that do not have easily alterable startup-sequence files. This includes many copy protected applications, games particularly.

4. Addmem has no mechanism for setting the priority of the memory that it adds. In the regular course of events, CHIP RAM is assigned a priority of -10, and autoconfigured FAST RAM is assigned a priority of zero. This insures that the FAST RAM will get allocated before CHIP RAM, unless CHIP RAM is explicitly indicated either in the hunks which the AmigaDOS loader reads, or in a program's memory allocation request.

Addmem assigns a priority of zero, the same as FAST RAM, to the memory it adds. This is reasonable in most cases: you can't Addmem CHIP RAM to the system and, because the data bus on the 68000 is only 16 bits wide, any memory added to the system is assumed to be 16-bit FAST RAM. The latter, of course, is *not* true of the LUCAS/FRANCES combination. The 32-bit wide FRANCES memory is clearly the fastest RAM in the system. We wanted a method to specify that the FRANCES memory have a higher priority than 16-bit FAST RAM.

5. Addmem also requires specifying the start and end addresses of the RAM being made available. This was awkward for our purposes since most people who build this memory board will probably not want to populate it with four megabytes of memory right away. It would be quite a nuisance for you to have to edit the startup-sequence files on all 42 of your boot disks everytime you wanted to add more RAM.

If you are running KickSsnetart out of FRANCES RAM, you would also have to correctly specify the range of RAM available as 256 Kb smaller than the full range of memory. This is not a big deal, but it would mean changing two places in the startup-sequence file instead of only one: the Addmem command, and the load KickStart into FRANCES RAM utility.

Since we were already going to have to write our own utility to configure the programable 8421A DRAM controller, copy KickStart to RAM and enable it, we decided to have it also add the memory and address the concerns listed above. Thus was born *AFM*, the FRANCES memory configuration utility. The name AFM stands for Add FRANCES Memory. This simple-to-use program is executed from your startup-sequence and deals with all of the situations listed above as best as possible for non-autoconfiguring memory.

The syntax for AFM is as follows:

```
AFM -mxxxxxx [-k] [-d] [-r] [-b]
```

Switches in square brackets are optional.

The **-m** is the ModeLoad switch and it is the only required switch. It is followed by a six digit hexadecimal number (24 bits) that specifies the programming of the DRAM controller. Don't panic – the FRANCES board documentation comes with the standard values for 16-, 18- and 20-MHz operation for all possible memory configurations. You will only have to pick the one that is right for your system. For brave souls who want to experiment with programming the DRAM controller, the description of what these bits do is provided in the FRANCES documentation.

The **-k** is the optional 32-bit KickStart switch. If it is specified, the top 256 Kb of FRANCES memory will not be made available to AmigaDOS. Instead, AFM will copy KickStart from the Amiga 1000 WCS (Writable Control Store) RAM to this area, and then enable the RAM KickStart feature of the FRANCES board by asserting ERKS. Once this is done, KickStart will run from 32-bit wide FAST RAM, which can make a whale of a difference for many applications.

The **-d** is the optional debug switch. It can be used to display information about AFM's activities if you are having any problems or if you are just interested. It will report where it found the the FRANCES memory and how much of it, and so on.

The **-r** is the optional Resident switch, the key to one of the most powerful features of the AFM program. If it is specified, AFM spawns a little program of less than 300 bytes that is

hooked into the AmigaDOS resident module list. It is then executed by AmigaDOS at warm boot time, even before autoconfiguration of peripherals occurs.

This resident module remembers what parameters AFM was run with, and will subsequently configure the FRANCES board the same way every time the Amiga is booted, until the Amiga is powered off. In other words, the FRANCES board will act as if it were an autoconfigure memory board until power is removed. Once the resident module is added, you can boot your favourite copy-protected version of Raster-Blaster, and if it can use fast RAM, it will use the 32-bit FRANCES RAM.

Whenever AFM is run, it checks to see if the resident module is already present. If it is, AFM knows that the memory has already been configured, and it exits without further ado. Someone once said about the RAMB0: RAM disk that it "hangs on like grim death" until the system is powered down. AFM is like RAMB0: in this respect. If you want to experiment with different programming values for the DRAM controller, don't use this switch.

The **-b** is the optional boot switch. Using this switch only makes sense in conjunction with the **-r** switch. If **-b** is specified, AFM will reboot the Amiga as soon as the resident module is spawned. The reason: the first time that you boot, the resident module will not be present, which will result in some stuff being loaded into CHIP RAM or 16-bit FAST RAM that should have gone into FRANCES RAM. These things will be loaded into FRANCES RAM the next time that you warm boot because the resident module will have already run. The **-b** switch will cause this second boot to happen immediately. This only takes a few seconds, and insures that you are running with as much stuff as possible loaded into FRANCES RAM right away. If you are using this switch, you really should have the AFM program as the first line of your startup-sequence file since anything you have done during the boot-up to this point will be lost.

The first time we saw this complete re-boot from software, we were quite startled; you have to experience it to understand.

The AFM program first programs the DRAM controller. It then checks location $40400000 and $00400000 for memory, in that order. If it does not find memory in either of those locations, it announces that no FRANCES memory was found and exits. Once the FRANCES memory has been found, AFM non-destructively checks the first byte of each megabyte boundary up to four megabytes to determine the amount of FRANCES memory available. If the RAM KickStart switch was selected, the size of the available memory is reduced accordingly, and KickStart is copied up to FRANCES RAM and enabled. AFM adds the available FRANCES memory to the AmigaDOS memory list with a priority of 10. If the resident module was requested, it is spawned and added to the resident list awaiting the next reboot. To the resident module is passed the location and extent of the FRANCES RAM, and whether KickStart is to be run from FRANCES memory. Finally, if the automatic reboot was requested, the Amiga is now rebooted; otherwise, AFM exits cleanly.

AFM has been written so that it will run whether or not there is an '020 or FRANCES memory in the system. If AFM finds that there is no '020, it exits with a return value of 5. This value can be checked in the startup-sequence script and conditional action taken. This is useful for those of you who have done Evan Sidorak's modification to the LUCAS board that allows you to switch between the 68020 and the 68000 (with a reboot, of course). The 32-bit FRANCES memory will not be available when using the 68000; by checking the return code of AFM, it is possible to write a startup-sequence script that will work for both 68000 and 68020 boots.

If AFM does find an '020 in the system, but cannot find the FRANCES memory, it exits with a return value of 10. This is taken to be the more serious failure because it indicates that the FRANCES memory may have failed.

We believe that this scheme gives the startup-sequence programmer sufficient flexibility to deal with any LUCAS/FRANCES situation.

Benchmarks

The following benchmarks give an indication of the raw computing performance of a stock A1000, an A2500 with A2626 card running KickStart in 32-bit space, and the LUCAS-FRANCES combination running at 16 and 20 MHz. These are the same bench marks that I used in the LUCAS article. The results will vary slightly each time you run them.

You should bear in mind that these benchmarks do not take into account hard disk transfer speed. To give you an idea of how this affects the system, Eric compiled his *Handshake* program once the LUCAS & FRANCES was installed. The hard disk light came on and just stayed on until the compile and link was complete. Guess where the bottleneck is now?

Where raw computing power really helps is with applications like ray-tracing. I find that, with the LUCAS-FRANCES installed, I am seeing my images rendered 2 1/3 times faster than with just the LUCAS board.

Benchmarks

	Whetstone	Savage	Calcpi	Float
Stock A1000	24	23842	4.87	286.1
A2500/2620 (K32)	247	444	23.75	58.1
LUCAS-FRANCES (16 MHz, K32)	277	402	26.18	54.8
LUCAS-FRANCES (20 MHz, K32)	295	352	29.46	48.0
UNITS	KWhets/sec	50*sec	Kflops/sec.	seconds, 256000 i

(K32 means KickStart running in 32-bit space)

Conclusion

There is much more information available on the disk that comes with the bare FRANCES board. If you have questions, Eric can be contacted on BIX as **ehaberfellner** or through USENET at **becker!haberfellner!eric**; and I can be reached on BIX as **anakin.1** or via USENET at **utgpu!anakin**.

If you're interested in acquiring either a bare LUCAS or FRANCES board and PALs, send a cheque or international money order to:

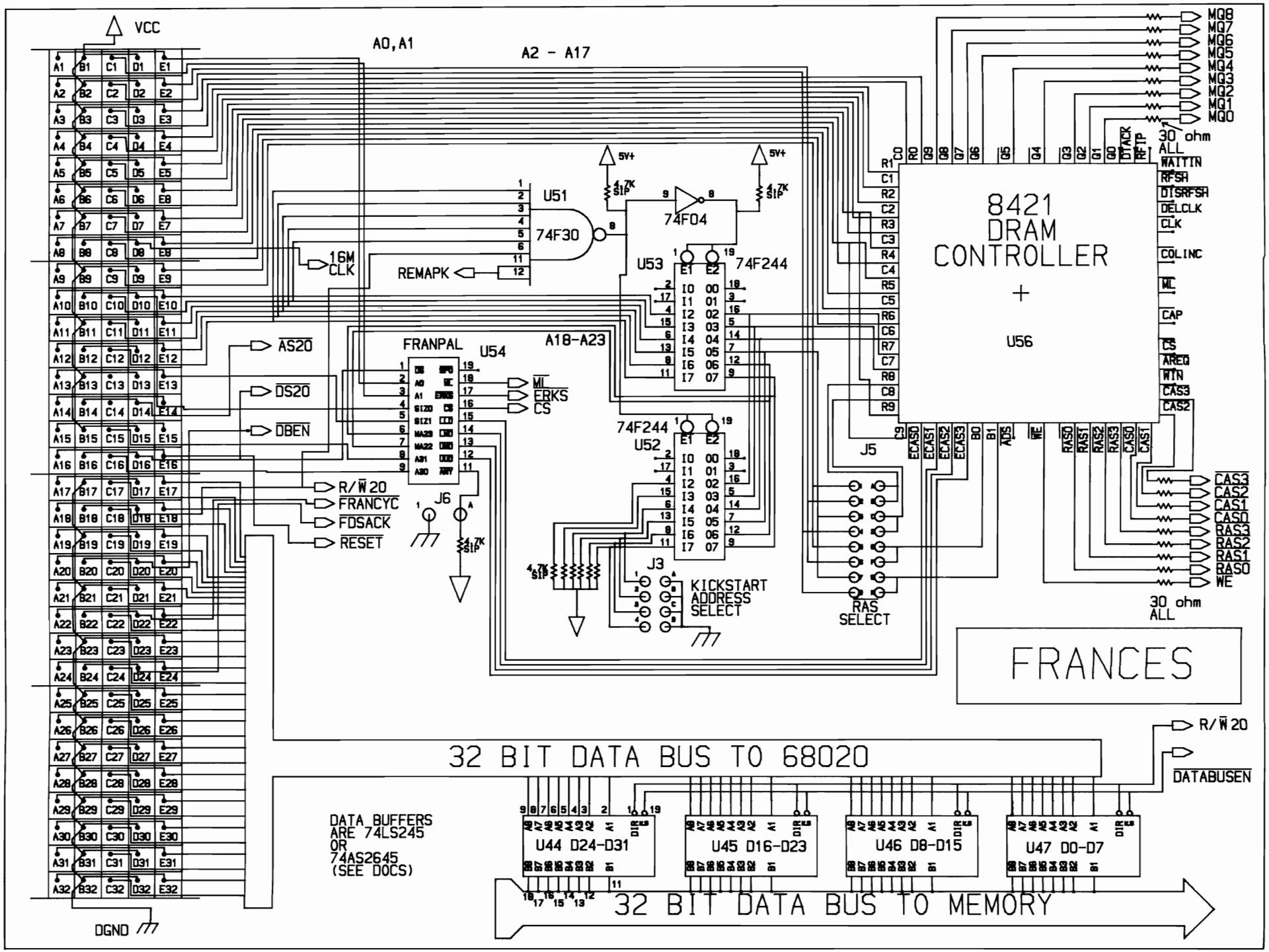
The LUCAS Project
c/o Brad Fowles
RR #5, Caledon East
Ontario, Canada.
L0N 1E0

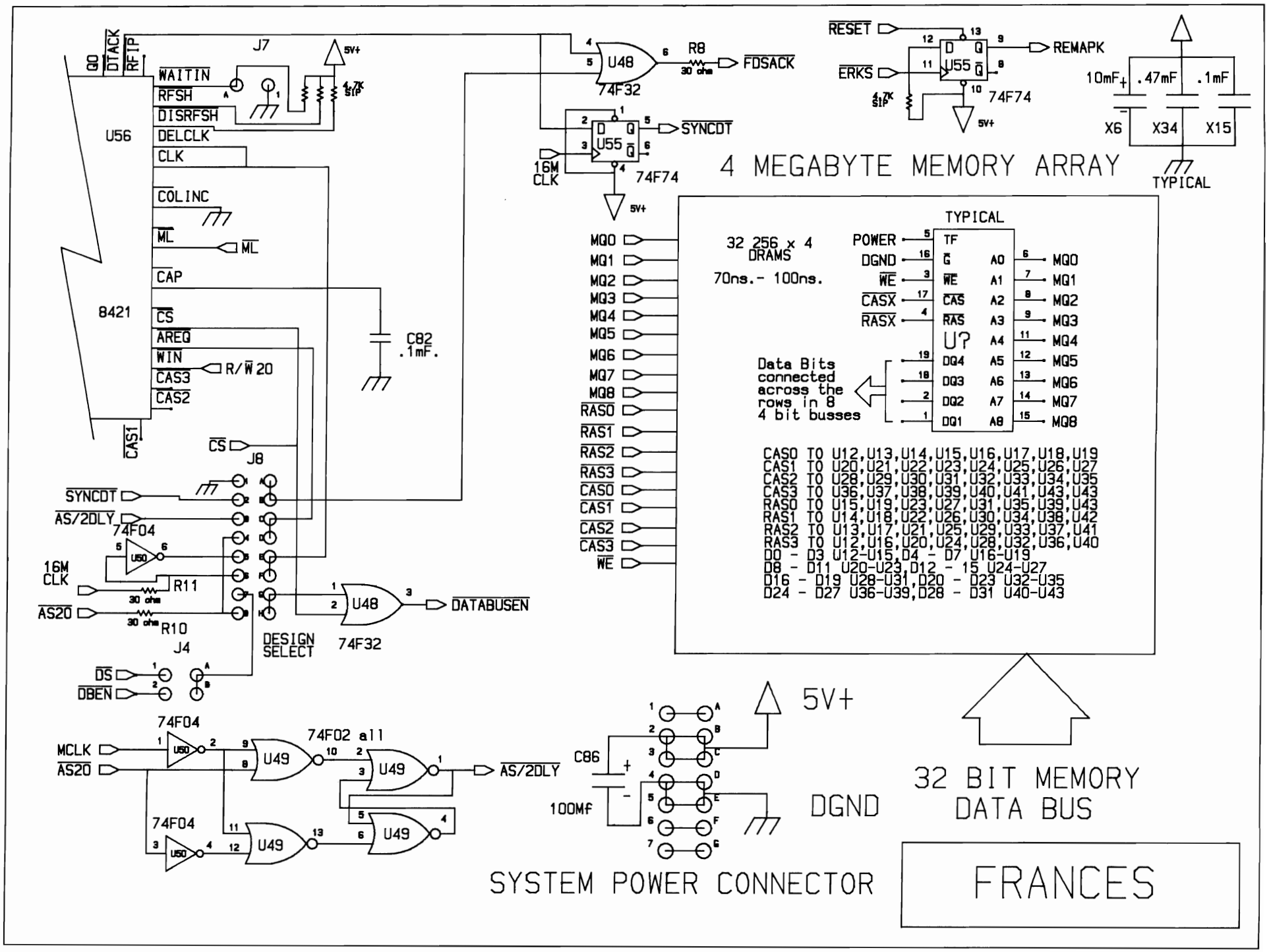
LUCAS, 4 pals, and documentation disk: $75.00 US
FRANCES 2 pals, and documentation disk: $75.00 US

Don't forget to specify the type of 96-pin DIN connector you used.

PAL equations:

```
PARTNO      U54 ;
NAME        FRANCES3 ;
REV         03 ;
DATE        March 3rd 1989;
DESIGNER    Brad Fowles ;
COMPANY     Anakin  ;
ASSEMBLY    Frances ;
LOCATION    U54 ;

/* PAL1618B2 */
/* PAL DESIGN SPECIFICATION */
/* 68020-68881 /68000 AMIGA INTERFACE */

PIN 1 = DS ;
PIN 2 = A0 ;
PIN 3 = A1 ;
PIN 4 = SIZ0 ;
PIN 5 = SIZ1 ;
PIN 6 = MA23 ;
PIN 7 = MA22 ;
PIN 8 = A31 ;
PIN 9 = A30 ;
PIN 11 = AMY ;
PIN 12 = UUD ;
PIN 13 = UMD ;
PIN 14 = LMD ;
PIN 15 = LLD ;
PIN 16 = CS ;
PIN 17 = ERKS ;
PIN 18 = ML ;
PIN 19 = SPO1 ;

!UUD =   !A0   & !A1   & !DS;

!UMD =   !SIZ0 & !A1   & !DS
       # A0    & !A1   & !DS
       # SIZ1  & !A1   & !DS;

!LMD =   !A0   & A1    & !DS
       # !A1   & !SIZ0 & !SIZ1 & !DS
       # SIZ0  & SIZ1  & !A1   & !DS
       # !SIZ0 & !A1   & A0    & !DS;

!LLD =   A0    & SIZ0  & SIZ1  & !DS
       # !SIZ0 & !SIZ1 & !DS
       # A0    & A1    & !DS
       # A1    & SIZ1  & !DS;

!CS  =  !MA23 & MA22  & !A31  & !A30  & AMY
     #  !MA23 & MA22  & !A31  & A30   & !AMY;

!ML  =  A31  & !A30  & MA23 & !MA22 & !DS ;

!ERKS =   A31 & !A30 & !MA23 & MA22 ;

/*
DESCRIPTION: BYTE WRITE DECODE FOR *CAS GENERATION AND SYSTEM DECODE
*/


PARTNO      U7 ;
NAME        NEWU7 ;
REV         01 ;
DATE        MARCH 3RD, 1989 ;
DESIGNER    Brad Fowles ;
COMPANY     Anakin  ;
ASSEMBLY    Lucas & Frances ;
LOCATION    U7 ;

/* PAL1618B2 */
/* PAL DESIGN SPECIFICATION */
/* 68020-68881 /68000 AMIGA INTERFACE */

PIN 1 = HIGHZ ;
PIN 2 = DS20DLY ;
PIN 3 = A0 ;
PIN 4 = SIZ0 ;
PIN 5 = SIZ1 ;
PIN 6 = AS20DLY ;
PIN 7 = CPCS ;
PIN 8 = CPDSACK0 ;
PIN 9 = FDSACK ;
PIN 11 = SYSDSACK1 ;
PIN 12 = DSACK0   ;
PIN 13 = FRANCYC ;
PIN 14 = DSACK1 ;
PIN 15 = CPDSACK1 ;
PIN 16 = AS00BUF ;
PIN 17 = AS00 ;
PIN 18 = LDS ;
PIN 19 = UDS ;

AS00.OE = HIGHZ & FRANCYC ;
!AS00   = (CPCS )& (!AS20DLY );

AS00BUF.OE = HIGHZ &FRANCYC;
!AS00BUF   = (CPCS )& (!AS20DLY );

UDS.OE = HIGHZ & FRANCYC;
!UDS  = (!DS20DLY )& (!A0 ) & (CPCS) ;

LDS.OE = HIGHZ & FRANCYC ;
!LDS  = ( !DS20DLY ) & ( SIZ1 ) & (CPCS) #
        ( !DS20DLY ) & (!SIZ0 ) & (CPCS) #
        ( !DS20DLY ) & ( A0 ) & (CPCS) ;

!DSACK1 = (!FDSACK )& (!AS20DLY )#
          (!CPDSACK1 )& (!AS20DLY )#
          (!SYSDSACK1)& (!AS20DLY ) ;

!DSACK0 = (!FDSACK )& (!AS20DLY )#
          (!CPDSACK0 )& (!AS20DLY ) ;

/*
DESCRIPTION: ADDRESS STROBE, UPPER AND LOWER DATA STROBE AND FINAL DSACKX
             GENERATION
*/
```


VCC
A0,A1
A2 - A17
MQ8
MQ7
MQ6
MQ5
MQ4
MQ3
MQ2
MQ1
MQ0
30 ohm ALL
U51
74F30
74F04
5V+
4.7K SIP
16M CLK
REMAPK
U53
74F244
8421 DRAM CONTROLLER
U56
WAITIN
RFSH
DISRFSH
DELCLK
CLK
COLINC
ML
CAP
CS
AREQ
WIN
CAS3
CAS2
FRANPAL
U54
A18-A23
AS20
DS20
DBEN
ML
ERKS
CS
R/W20
FRANCYC
FDSACK
RESET
J6
74F244
U52
J5
J3
KICKSTART ADDRESS SELECT
RAS SELECT
CAS3
CAS2
CAS1
CAS0
RAS3
RAS2
RAS1
RAS0
WE
30 ohm ALL
FRANCES
R/W20
DATABUSEN
32 BIT DATA BUS TO 68020
DATA BUFFERS ARE 74LS245 OR 74AS2645 (SEE DOCS)
U44 D24-D31
U45 D16-D23
U46 D8-D15
U47 D0-D7
32 BIT DATA BUS TO MEMORY
DGND


4 MEGABYTE MEMORY ARRAY
32 BIT MEMORY DATA BUS
SYSTEM POWER CONNECTOR
FRANCES
U56
8421
J7
J8
J4
U48
74F32
U55
74F74
R8
30 ohm
FDSACK
SYNCDT
16M CLK
REMAPK
RESET
ERKS
4.7K SIP
5V+
10mF
.47mF
.1mF
X6
X34
X15
TYPICAL
C82 .1mF.
R/W 20
74F04
R11
R10
DATABUSEN
DESIGN SELECT
MCLK
AS20
AS/2DLY
74F02 all
C86
100MF
DGND
32 256 x 4 DRAMS
70ns.- 100ns.
POWER
DGND
WE
CASX
RASX
U?
Data Bits connected across the rows in 8 4 bit busses
CAS0 TO U12,U13,U14,U15,U16,U17,U18,U19
CAS1 TO U20,U21,U22,U23,U24,U25,U26,U27
CAS2 TO U28,U29,U30,U31,U32,U33,U34,U35
CAS3 TO U36,U37,U38,U39,U40,U41,U43,U43
RAS0 TO U15,U19,U23,U27,U31,U35,U39,U43
RAS1 TO U14,U18,U22,U26,U30,U34,U38,U42
RAS2 TO U13,U17,U21,U25,U29,U33,U37,U41
RAS3 TO U12,U16,U20,U24,U28,U32,U36,U40
D0 - D3 U12-U15,D4 - D7 U16-U19
D8 - D11 U20-U23,D12 - 15 U24-U27
D16 - D19 U28-U31,D20 - D23 U32-U35
D24 - D27 U36-U39,D28 - D31 U40-U43

Dynamics For Animators, Part 2

Elastic 3D objects in motion

by Cathryn and Eric Graham

Eric and Cathryn have been involved with three dimensional modeling on the Amiga for several years, producing the Juggler Demo, Sculpt 3D and Sculpt Animate, as well as magazine articles, and the Ray Tracing Newsletters. They live in a high mountain valley in Colorado. They can be reached at P.O. Box 8298, Durango, Colorado 81301. A disk containing the source code in this article, an executable version and some dynamic animations is available for $6.00, including postage.

In part 1 of this article, we introduced you to dynamics and showed you that the motion of an object depends upon the forces that are applied to it. The exact details of the motion can be calculated by solving a system of differential equations; we showed you a simple way to solve the equations.

In this article we will apply these methods to calculate the motion of elastic three dimensional objects, and present a program that will generate data files of the resulting motion for use in three dimensional rendering and animation programs.

Initial values

The method we will employ is called the initial value approach. We will specify the position and velocity of objects at an initial time. Then we solve the equations to find the position of the objects at later times.

In the article *Algorithmic solid model generation* in *TransAmi* Volume 1 Issue 5, we showed how solid objects could be represented in a computer, and how to output the representation in a form compatible with many solid modeling programs. Objects were built from triangular faces. Each face was described by the position of its corners, called vertices. The sides of the triangles were called edges. The connections between edges, faces and vertices were described by a set of C-language data structures. Let us extend these structures to include some fields that we will need to treat the dynamics of solid objects:

```
struct vertex
        {struct vertex *next;
         double pos[3];
         struct edgelist *edgelist;
         long int index;
         double mass;
         double velocity[3];
         double force[3];
         double ws[48];     /* working storage */
        };

struct edgelist
        {struct edgelist *next;
         struct edge *edge;
        };

struct forcelaw
        {double origlength;
         double restore;
         double damp;
        };

struct edge
        {struct edge *next;
         struct vertex *vertices[2];
         struct forcelaw forcelaw;
        };

struct object
        {struct vertex *vertices;
         struct edge *edges;
         struct face *faces;
         double gravity;
         double groundcoef;
         double restitution;
        };


struct face
        {struct face *next;
         struct vertex *vertices[3];
         long int properties;
        };
```

We will be describing the new fields in the course of this article. Our first problem is to read a Sculpt 3-D compatible scene file, so that we can animate arbitrary objects that have been built with a solid modelling program.

A scene file is in the IFF format and can contain many different types of chunks. The mechanics of reading and writing IFF files are described in the ROM Kernel Manual as well as in documents that can be found in most Amiga public domain disk collections. Listing 1 is an IFF read routine that extracts all the information that we will need. The functions *newvertex, newedge, newface* and *newedgelist* were presented in the earlier article.

Half of our problem of gathering initial data is now solved. We have the positions of the vertices that make up our solid objects. How can we provide the initial velocities?

We can specify velocities using our solid modeling program, if we adopt the following procedure: for each vertex of the model add a second vertex, displaced in a direction to represent the initial velocity of the vertex. Link the two vertices with an edge, so that there can be no ambiguity about which vertex belongs to which.

To extract the velocity information, we only have to look at all the vertices that have but one edge. Follow the edge back to the vertex at the other end, and save the velocity. To clean things up, we should then delete the vertices and edges that are used to represent velocity. The functions *getvelocity, trimedgelist, trimedge* and *trimvertex* in Listing 2 show how it is done.

The velocity information is stored as a part of the vertex data structure. The function *getvelocity* is called with a parameter *dt* that represents the time difference that is used to convert a displacement into a velocity.

At this point we should digress and discuss the matter of units. We have scaled the fixed point representation used by scene files by a factor of one hundred thousand to come up with conveniently sized floating point values. In Sculpt 3-D, if you employ a coordinate system with four decimal places, the numbers displayed by the coordinate window will match the values used by our dynamics program. In dynamics you are concerned with units for distance, mass and time. Choose a unit of distance that is appropriate for the kind of objects that you are dealing with; feet, for example. The second is a good unit of time, if you are dealing with everyday motions. The unit of mass is less important, because it cancels out in all our calculations. A little later we will give each vertex the same mass, a value of unity.

Forces

Forces are what make the world move. It is here that you can use all of your creativity. We will deal with just three forces: gravity, ground impact forces, and elastic forces that hold objects together.

The forces that deal with gravity and the interaction of an object with an almost solid ground were described in part 1 of this article. If we did not provide cohesive forces, the cloud of vertices that comprise our solid objects would quickly disperse and the objects would become unrecognizable. In nature, this is accomplished by the forces between atoms and molecules. We can model this by attributing a tension to each edge.

A good model for the edge force is provided by the spring and shock absorber in an automobile suspension. A spring has a natural length and exerts forces to resist either compression or extension. The damping action of a shock absorber keeps vibrations from persisting too long.

The data structure *forcelaw* provides the parameters needed for each edge. The parameters for gravity and ground forces are contained in the object data structures. The *setupforces* function in Listing 2 sets up these values.

The function takes a number of parameters. The parameter *trestoring* is the amount of time (timescale) that elastic forces take to act. The shorter the timescale, the more rigid the object will be. An object will take a few of these timescales to oscillate once, so a realistic time scale for an everyday solid object might only be a few hundred microseconds. It is advisable not to make objects too stiff, as the computing time to solve the differential equations increases for stiff objects. More interesting animations can be made when the eye has time to see the objects deform.

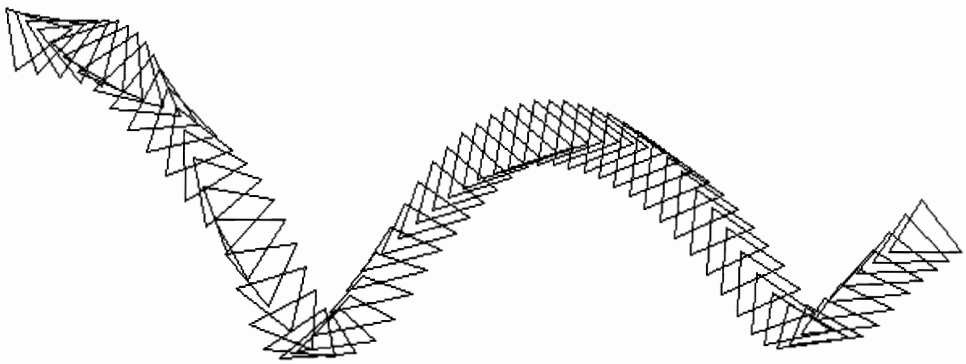
The parameter *tdamping* is the timescale for vibration to die down. If it is much larger than the elastic timescale, then objects oscillate many times; otherwise vibration will quickly dissipate.

The parameter *tground* determines timescale for ground impact forces to respond. As described in part 1, we treat the ground as an elastic membrane, so objects can pass below ground level before bouncing back. The shorter the timescale, the less objects will penetrate below the ground.

The last parameter sets the gravitational acceleration. A value of 32 feet per second per second is appropriate for the surface of our planet.

The function *distance* calculates the distance between two points, and *calcforce* calculates forces on each vertex. For each vertex, we calculate the force of gravity, and possibly the ground impact force. Finally, we calculate the cohesive forces of each edge using the *calctension* routine. In this function, we examine each edge and calculate a tension based on how much the edge has been stretched or shrunk from its orignal length. The damping force is derived from the rate at which the edge is currently shrinking or expanding. The force is applied equally, but with opposite direction to the two vertices at the endpoints of the edge. This is Newton's law, "Every action has an equal and opposite reaction".

The functions *dot, vsprod* and *vecsub* perform a vector dot product, a vector-scalar product and a vector subtraction, respectively. They were given in the article on solid model generation.



Simple dynamic animation with spin

The illustration shows a very simple dynamic animation. A single triangular face has been thrown horizontally and given a spin. It follows a curved trajectory, still spinning, and bounces off the ground. When it hits the ground, it deforms and oscillates as it springs back into the air. When animated, it seems to behave in a most realistic way.

Much more complicated animations are possible, but are difficult to represent on paper; they must be seen as animations to be appreciated.

Some words of caution are required when you construct objects for dynamic animation. Imagine a circle of linked edges; although it will resist stretching, it has no more rigidity than a loop of string. To make an elastic disk, you have to triangulate the interior. A three dimensional model of a human head, for example, may not be very rigid; it will behave like a rubber mask. You will have to add internal edges as cross-bracing to make it behave as a solid body. Considerable experimentation with the timescales may be needed in order to get just the effects that you desire. Avoid making the timescales too short, or the computational time will lengthen.

Several enhancements suggest themselves. The treatment of the ground could include horizontal resistance. A ball hitting such a ground at an oblique angle would then spin; or a spinning ball would bounce back in that unexpected way that confounds novice tennis players.

With ground friction, you could also set up an oblique spinning top and observe the effect of gyroscopic precession.

A more complicated task would be to include the interaction between disconnected objects. Additional close-range forces could be applied that repel one object from another. Although this would not preclude an object from passing through another, the damage to both objects would be apparent. Imagine firing a bullet into an apple. If the velocity were low enough, the bullet would bounce back. At higher speed the bullet could pass through, and the apple would experience a wave of disruption associated with the passage of the projectile. Because we are handling dynamics in a realistic manner, the apple would also be knocked back and the bullet would be slowed down.

There really is no limit to the kind of dynamic simulations that can be performed in order to generate a realistic animation. We look forward to seeing the results of your imagination.

Solving the equations

If you have an object comprising a hundred vertices, then you will have to solve a 600 degree differential system! Take heart - the adaptive Runge-Kutta-Merson technique described in part 1 can easily be modified to solve such systems, although the computation time may not be short.

In the earlier article, a general equation solving system was presented. The numerical method requires quite a lot of temporary storage, and rather than allocate it in a huge array, we choose to distribute it among the vertices, so the structure of the equation solver changes, although the principle remains the same. See the function *arkmstep* in Listing 2.

The method generates an error estimate, and controls the step size in an attempt to satisfy the accuracy that you request. The basic equation solving step is done with the function *rkmstep*. *Arkmstep* employs an auxiliary function, *copyws*, to move data within the temporary storage space and to the primary fields *pos* and *velocity*. Finally, the function must be supplied with the rates at which the dependent variables (position and velocity) change. This is accomplished by the *derivs* function.

Dynamic animations

A complete program must read in a scene file, extract the velocity information, set up 'forcelaw' parameters, perform a step-by-step solution of the equations and output the results in the form of a series of *Sculpt 3-D* compatible scene files. The *main* routine in Listing 2 shows how this could be done. This program is set up to read in a file called 'test.scene' and output files 'test.0.scene', 'test.1.scene', etc. They could then be directly used by a program like *Sculpt Animate 4D*.

Listing 1: IFF reader for scene files

```
#define MAKEID(a,b,c,d) (((long int)a)<<24|((long int)b)<<16|\
                         ((long int)c)<<8|((long int)d))

#define FORM    MAKEID('F','O','R','M')
#define SC3D    MAKEID('S','C','3','D')
#define VERT    MAKEID('V','E','R','T')
#define EDGE    MAKEID('E','D','G','E')
#define FACE    MAKEID('F','A','C','E')


struct evertex
        {long int pos[3];
        };

struct eedge
        {long int evertexi[2];
        };

 struct eface
        {long int evertexi[3];
         long int properties;
        };


rdscene(name,object)
char *name;
struct object *object;
{FILE *fopen(),*file;
 long int ptr,ptr2,ckid,ckidsize,len;
 double dpos[3];
 struct evertex evertex;
 struct eedge eedge;
 struct eface eface;
 struct vertex *vertex,**vertindex;
 struct edge *edge;
 struct face *face;
 int i,k,n,elesize;
 char *calloc();

 object->vertices=0L; object->edges=0L; object->faces=0L;
 file=fopen(name,"r");
 if (!file) {printf("Unable to open file\n"); exit(0);}

 vertindex=0L;
 ptr=0L;

 for (;;)
   {if (fseek(file,ptr,0))
        {fse:printf("file seek error\n"); exit(0);}
    if (fread(&ckid,4,1,file) != 1)
        {fre:printf("file read error\n"); exit(0);}
    if (fread(&ckidsize,4,1,file) != 1) goto fre;
    ptr2=ptr+12;
    if (ckidsize&1) ++ckidsize;
    ptr+=8+ckidsize;
    switch(ckid)
      {case FORM:
        if (fread(&ckid,4,1,file) != 1) goto fre;
        if (ckid != SC3D) break;
        while (ptr2 < ptr)
          {if (fseek(file,ptr2,0)) goto fse;
           if (fread(&ckid,4,1,file) != 1) goto fre;
           if (fread(&ckidsize,4,1,file) != 1) goto fre;
           len=ckidsize;
           if (ckidsize&1) ++ckidsize;
           ptr2+=ckidsize+8;
           switch (ckid)
             {case VERT:
                n=len/(elesize=sizeof(struct evertex));
                vertindex=(struct vertex **)calloc(n,sizeof(struct vertex *));
                if (!vertindex)
                      {printf("not enough memory\n"); exit(0);}
                for (i=0; i<n; ++i)
                 {if (fread(&evertex,elesize,1,file) != 1) goto fre;
                  for (k=0; k<3; ++k) dpos[k] = 0.00001 * evertex.pos[k];
                  vertindex[i]=object->vertices=
                     newvertex(dpos[0],dpos[1],dpos[2],object->vertices);
                 }
                break;
             case EDGE:
                n=len/(elesize=sizeof(struct eedge));

                while (n--)
                 {if (fread(&eedge,elesize,1,file) != 1) goto fre;
                  object->edges=
                      newedge(eedge.evertexi[0],
                         eedge.evertexi[1],object->edges);
                 }
                break;

             case FACE:
                n=len/(elesize=sizeof(struct eface));
                while (n--)
                 {if (fread(&eface,elesize,1,file) != 1) goto fre;
                  object->faces=newface(
                       eface.evertexi[0],
                       eface.evertexi[1],
                       eface.evertexi[2],
                       eface.properties,object->faces);
                 }
                break;

             default:
                break;
             }
          }

        for (edge=object->edges; edge; edge=edge->next)
            {for (k=0; k<2; ++k)
                {vertex=vertindex[(long int)(edge->vertices[k])];
                 edge->vertices[k]=vertex;
                 vertex->edgelist=newedgelist(edge,vertex->edgelist);
                }
            }

        for (face=object->faces; face; face=face->next)
            {for (k=0; k<3; ++k)
                {vertex=vertindex[(long int)(face->vertices[k])];
                 face->vertices[k]=vertex;
                }
            }

        if (vertindex) free(vertindex);
        fclose(file);
        return;
      }
   }
}
```

Listing 2: Assorted functions for 3D dynamics

```
getvelocity(object,dt)
struct object *object;
double dt;
{struct vertex *v1,*v2;
 struct edgelist *el;
 struct edge *e;
 int k;

 for (v1=object->vertices; v1; v1=v1->next)
    if (el=v1->edgelist)
        if (!(el->next))
            {e=el->edge;
             if ((v2=e->vertices[0]) == v1) v2=e->vertices[1];
             for (k=0; k<3; ++k)
                v2->velocity[k]=(v1->pos[k]-v2->pos[k])/dt;
             trimedgelist(&(v1->edgelist),e);
             trimedgelist(&(v2->edgelist),e);
             trimedge(object,e);
             trimvertex(object,v1);
            }
}


trimedgelist(pel,e)
struct edgelist **pel;
struct edge *e;
/*      Remove an edge from a list of edges */
```

```
{struct edgelist *el,*el0;

 for (el=*pel,el0=0L; el; el=el->next)
    if (el->edge == e)  /* found it? */
        {if (el0) el0->next=el->next;
         else *pel=el->next;
         free(el);
         return;
        }
    else el0=el;
}


trimedge(object,e)
struct object *object;
struct edge *e;
{struct edge *e0,*el;

 for (e0=0L,el=object->edges; el; el=el->next)
    if (el == e) /* found it */
        {if (e0) e0->next=el->next;
         else object->edges=el->next;
         free(el);
         return;
        }
    else e0=el;
}


trimvertex(object,v)
struct object *object;
struct vertex *v;
{struct vertex *v0,*v1;

 for (v0=0L,v1=object->vertices; v1; v1=v1->next)
    if (v1 == v) /* found it */
        {if (v0) v0->next=v1->next;
         else object->vertices=v1->next;
         free(v1);
         return;
        }
    else v0=v1;
}


setupforces(object,trestoring,tdamping,tground,g)
struct object *object;
double trestoring,tdamping,tground,g;
{struct edge *edge;
 struct vertex *v0,*v1,*v;
 double *p0,*p1,distance(),totmass;

 for (edge=object->edges; edge; edge=edge->next)
    {p0=(v0=edge->vertices[0])->pos;
     p1=(v1=edge->vertices[1])->pos;
     edge->forcelaw.origlength=distance(p0,p1);
     totmass=v0->mass+v1->mass;
     edge->forcelaw.restore=totmass/(trestoring*trestoring);
     edge->forcelaw.damp=totmass/tdamping;
    }

 for (totmass=0.0,v=object->vertices; v; v=v->next)
    totmass=totmass+v->mass;

 if (tground) object->groundcoef=totmass/(tground*tground);
 else object->groundcoef=0.0;
 object->gravity=g;
 object->restitution=0.8;
}


double distance(p0,p1)
double p0[3],p1[3];
{int k;
 double t,d,sqrt();

 for (t=0.0,k=0; k<3; ++k)
    {d=p0[k]-p1[k]; t=t+d*d;}
 return sqrt(t);
}


calcforce(object)
struct object *object;
{struct vertex *vertex,*v2;
 struct edge *edge;
 struct edgelist *el;
 double z,groundcoef,*force;

 for (vertex=object->vertices; vertex; vertex=vertex->next)
    {force=vertex->force;
     force[0]=force[1]=0.0;
     force[2]=-object->gravity*vertex->mass; /* gravity */
     if ((groundcoef=object->groundcoef) != 0.0)
        {z=vertex->pos[2];
         if (z < 0)  /* impact with the ground */
            {if (vertex->velocity[2] < 0)
                force[2]=force[2]-groundcoef*z;
             else
                force[2]=force[2]-groundcoef*object->restitution*z;
            }
        }
    }
 calctension(object);
}


calctension(object)
struct object *object;
{struct edge *edge;
 double *p0,*p1,distance(),e;
 struct forcelaw *fl;
 struct vertex *v0,*v1;
 double direction[3],relvel[3],tension,expansion,dot();
 int k;

 for (edge=object->edges; edge; edge=edge->next)
    {p0=(v0=edge->vertices[0])->pos;
     p1=(v1=edge->vertices[1])->pos;
     e=distance(p0,p1);
     fl=&(edge->forcelaw);
     tension=fl->restore*(e-fl->origlength);
     vecsub(direction,p1,p0);
     if (e != 0.) vsprod(direction,1.0/e,direction);
     vecsub(relvel,v1->velocity,v0->velocity);
     if (e != 0) expansion=dot(relvel,direction)/e;
     else expansion=0.0;
     tension=tension+fl->damp*expansion;
     for (k=0; k<3; ++k)
        {v0->force[k]=v0->force[k]+tension*direction[k];
         v1->force[k]=v1->force[k]-tension*direction[k];
        }
    }
}


arkmstep(object,t0,dt0,nt,e)
struct object *object;
double t0;              /* independent variable */
double dt0;             /* step size for t */
double e[];             /* maximum error tolerated */
int *nt;                /* pointer to subdivision */
{int i,j;
 double *y0,*toterr,*e2,dt,t,err,ratio;
 int n,iy0,itoterr,ie2,iy;
 struct vertex *v;

 n=6;
 iy0=4*n;
 itoterr=5*n;
 ie2=6*n;
 iy=7*n;

 if (*nt < 1) *nt=1;

 copyws(object,-1,iy0);
 copyws(object,-1,iy);

 tryagain:
 for (v=object->vertices; v; v=v->next)
   for (j=0; j<n; ++j)
        v->ws[itoterr+j]=0.0;
```

```
 dt=dt0/(*nt);
 t=t0;

 /* step out solution with *nt steps */
 for (i=0; i<(*nt); ++i)
     {rkmstep(object,iy,t,dt,n,ie2);
      t=t+dt;
      for (v=object->vertices; v; v=v->next)
        for (j=0; j<n; ++j)
           {err=v->ws[ie2+j]; if (err < 0.0) err=-err;
            v->ws[itoterr+j]+=err;
           }
     }

 /* is it good enough? */
 ratio=0.0;
 for (v=object->vertices; v; v=v->next)
   for (j=0; j<n; ++j)
     if (e[j] > 0.0)
        {err=v->ws[itoterr+j]/e[j];
         if (err > ratio) ratio=err;
        }
 if (*nt < 512 && ratio > 1.0)
    {*nt*=2;
     copyws(object,iy0,iy);
     goto tryagain;
    }

 /* is it too good? */
 if (*nt > 1 && ratio < 0.01) *nt/=2;
 copyws(object,iy,-1);
 return ratio < 1.0;
}


rkmstep(object,iy,t,dt,n,ie)
struct object *object;
int iy;                  /* index to dependent variables */
double t;                /* independent variable */
double dt;               /* step size for t */
int n;                   /* number of equations */
int ie;                      /* index to error estimate returned */
{int idy,ia,ida,ib,idb,ic,idc,idy1;
 double *dy,*a,*da,*b,*db,*c,*dc,*dy1,*y,*e;
 int i;
 struct vertex *v;

 idy=ie;
 ia=0;
 ida=n;
 ib=0;
 idb=2*n;
 ic=0;
 idc=n;
 idy1=3*n;

 derivs(object,idy,iy,t);
 for (v=object->vertices; v; v=v->next)
    {a=v->ws+ia; dy=v->ws+idy; y=v->ws+iy;
     for (i=0; i<n; ++i) a[i]=y[i]+dt*dy[i]/3.0;
    }

 derivs(object,ida,ia,t+dt/3.0);
 for (v=object->vertices; v; v=v->next)
    {b=v->ws+ib; dy=v->ws+idy; da=v->ws+ida; y=v->ws+iy;
     for (i=0; i<n; ++i) b[i]=y[i]+dt*(dy[i]+da[i])/6.0;
    }

 derivs(object,idb,ib,t+dt/3.0);
 for (v=object->vertices; v; v=v->next)
    {c=v->ws+ic; dy=v->ws+idy; db=v->ws+idb; y=v->ws+iy;
     for (i=0; i<n; ++i) c[i]=y[i]+dt*(0.125*dy[i]+0.375*db[i]);
    }

 derivs(object,idc,ic,t+0.5*dt);
 for (v=object->vertices; v; v=v->next)
    {dc=v->ws+idc; dy=v->ws+idy; db=v->ws+idb; y=v->ws+iy;
     for (i=0; i<n; ++i) y[i]=y[i]+dt*(0.5*dy[i]-1.5*db[i]+2.0*dc[i]);
    }

 derivs(object,idy1,iy,t+dt);
 for (v=object->vertices; v; v=v->next)
    {e=v->ws+ie; dy=v->ws+idy; db=v->ws+idb; dc=v->ws+idc; dy1=v->ws+idy1;
     for (i=0; i<n; ++i)
        e[i]=dt*(2.0*dy[i]+8.0*dc[i]-9.0*db[i]-dy1[i])/3.0;
    }
}


copyws(object,ifrom,ito)
struct object *object;
int ifrom,ito;
{int k;
 struct vertex *v;

 for (v=object->vertices; v; v=v->next)
    {if (ifrom < 0)
        for (k=0; k<3; ++k)
            {v->ws[ito+k]=v->pos[k];
             v->ws[ito+k+3]=v->velocity[k];
            }
     else if (ito < 0)
        for (k=0; k<3; ++k)
            {v->pos[k]=v->ws[ifrom+k];
             v->velocity[k]=v->ws[ifrom+k+3];
            }
     else for (k=0; k<6; ++k) v->ws[ito+k]=v->ws[ifrom+k];
    }
}


derivs(object,idy,iy,t)
struct object *object;
int idy,iy;
double t;
{struct vertex *v;
 int k;

 copyws(object,iy,-1);
 calcforce(object);

 for (v=object->vertices; v; v=v->next)
    {for (k=0; k<3; ++k)
        {v->ws[idy+k]=v->ws[iy+k+3];
         v->ws[idy+k+3]=v->force[k]/v->mass;
        }
    }
}


main()
{struct object object;
 double trestoring,tdamping,tground,dt,dtv,t,e[6],g;
 char buf[100];
 int i,nt;

 printf("trestoring?\n"); scanf("%lf",&trestoring);
 printf("tdamping?\n"); scanf("%lf",&tdamping);
 printf("tground?\n"); scanf("%lf",&tground);
 printf("dtv?\n"); scanf("%lf",&dtv);
 printf("g?\n"); scanf("%lf",&g);

 dt=1.0/30.0;   /* two jiffies */
 t=0.0;
 for (i=0; i<6; ++i) e[i]=0.01;
 nt=2;

 rdscene("test.scene",&object);
 getvelocity(&object,dtv);
 wrscene("test.0.scene",&object);
 setupforces(&object,trestoring,tdamping,tground,g);

 for (i=1; i<=60; ++i)
    {arkmstep(&object,t,dt,&nt,e);
     sprintf(buf,"test.%d.scene",i);
     wrscene(buf,&object);
     t=t+dt;
    }
}
```

Toy: A Small Compiler For The Amiga

Part 2: Data structures, symbol table and tokenizer

by Chris Gray

Chris Gray is a programmer from Edmonton, Alberta. He writes: "I currently working for Myrias Research Corporation, which is building an easy-to-use parallel supercomputer. Our first regular customer machine was shipped May 1989, and we look forward to giving the big boys like Cray and NEC a run for their money. I fell in love with the Amiga when it was first announced, and bought one of the first in the city. Since then I've been Amiga programming steadily. My major software releases, both shareware, are the Draco compiler, along with its accompanying libraries and tools, and Amiga Empire."

In part 1 of this series, I described the Toy language and briefly outlined the Toy compiler. In this instalment I begin the detailed description of the compiler and its operation. With this article, as well as with future ones in the series, are accompanying listings of the compiler. The files listed are shown in their entirety, but some files are only described, and not listed. These files are available on the disk for this issue.

This article has three main portions. The first describes the data structures that the compiler uses internally. This is done using the compiler's main include file, 'toy.g' (see Listing 1 at the end of the article). Not all of the structures will be fully described, since they are not encountered until later in the compiler. The second portion of the article will describe the symbol table routines used by the compiler. These are used for keeping track of the identifiers (variable, procedures, and so on) declared in the program. The final portion describes the tokenizer, the portion of the compiler that breaks up the input text stream into a stream of larger tokens that the compiler then parses.

Data structures

The 'toy.g' file is included with all source files of the compiler. This means that all of the types and values declared in it are available throughout the compiler. On a somewhat larger project, it would probably be better to split the information up so that most types are known only to those portions of the project that require them ('information hiding'). The second include file used by the compiler is also included in all source files. It is 'externs.g', and it includes declarations of all functions that are used outside of the file that defines them. This file is not listed here since it is straightforward. This usage ensures that all such functions are used properly, since all uses are made with the same declaration in effect, and the file that defines the function also has that declaration in effect, so the compiler can check the declaration against the actual definition. This same technique can also be used with ANSI C. Again, on a larger project, it may be better to split up the file.

The first type defined in 'toy.g' is one that we will see a lot of in this and the next article. It is an enumeration type listing all of the legal token values that the tokenizer will generate. Some of the tokens are accompanied by additional information; for instance, *tk_number* needs the value of the number. Others, such as *tk_proc* require no additional information. For some compilers, it is useful to also define a structure that can represent any token completely. That way, token values can be readily passed around. There was no need for this in Toy, however.

The second type, *ResultType_t* (the company I work for has a programming convention in which all type names end in '_t', so I have been using that convention in my recent non-work programming), is a much smaller enumeration of the types that the compiler knows about. Type *rt_error* is used internally to cut down on the error messages issued in response to a single error. Most compilers have more complex types and need a more complex type structure.

Next comes type *SymbolKind_t*, yet another enumeration, this time of all of the kinds of symbols that we can store in our symbol table. Note that Toy (like Draco) stores the language's reserved words in the main symbol table. The closely related type *SymbolEntry_t* describes the structure of each symbol table entry. It contains a pointer to the name of the symbol, the length of that string (including the terminating **\e**), a value of type *SymbolKind_t* saying what kind of symbol this is, the type of the symbol, and a couple of fields used only for some kinds of symbols. The length of the name need not be stored, but doing so allows slightly quicker equality checks in some cases, and avoids recomputing the length when the symbol is freed.

The final type, *RelocCode_t* , I will not describe here, except to say that it has to do with code generation and emitting the final object file.

Anyone who has worked with a real compiler will immediately notice that there are no data structures to contain any

compiler state information (except the symbol table of course), or to hold any kind of representation of a tokenized or parsed program. As mentioned in the first article, the Toy compiler doesn't do any of that: the parsing routines call the tokenizer directly, and also call the code generation routines directly, so no intermediate representations are needed. This is true to a lesser extent of the Draco compiler also. It does maintain state information about register usage, and a small buffer of instructions for the peephole optimizer, however.

The symbol table

Most compilers will have more than one symbol table. For example, a Pascal compiler will often create a new symbol table for each BEGIN-END block to contain the symbols declared in that block. Thus, when it is trying to find the meaning of a symbol, it starts with the table for the most local block and works outward until it finds a definition. Another way of handling this is to have a single table, but to stack the definitions in that table, so that earlier ones are hidden from view by more local definitions. A rule with both Toy and Draco is that such hiding is not allowed: there can be only one definition of a given symbol at any given time. With this rule, a single symbol table is all that is needed. Also, since Toy's special tokens like **if** are 'reserved words' as opposed to 'keywords', the user cannot declare identifiers that look like them. Thus, they too can be in the single symbol table with no conflict. The only requirement on the single table is that it be possible to remove local variables from it at the end of each procedure.

Listing 2, 'symbol.d', is the symbol table code for Toy. The table itself is a 1023-slot (1023 being a nice prime number) hash table. The first routine, *symbolInit,* simply initializes all slots to *sk_empty*, indicating that they are unused. It is called by the per-file start-up code. Global variable *EntryCount* is used to count the number of entries in the table. If that limit is reached, the Toy compiler will abort. A fairly easy alternative would be to use a dynamically allocated table, and, if it gets too full, allocate a new one twice as big, and re-enter all symbols into the new table. The variable *se* is declared as a pointer to *SymbolEntry_t*. It scans along the elements of the array, which are **sizeof(SymbolEntry_t)** bytes apart.

The *memFree* routine is defined in the system interface portion of the compiler. That portion is not shown here, but the part needed to test the tokenizer is available on disk. *memFree* simply calls the Amiga Exec routine *FreeMem* to free the memory. Other routines from that system interface are used later in the symbol table code, and also in the tokenizer.

symbolTerm is the corresponding routine to clean up the hash table. Specifically, it frees the strings associated with all used entries. Note that it uses the stored *se_length* field as the length in characters of the string. The multiplication by **sizeof(char)** is not really necessary since characters are one byte long. I am, however, trying to prepare myself for the eventuality of multibyte characters, such as the ANSI C people are discussing, so I try to include the required scaling where appropriate.

symbolPurge performs the function mentioned earlier: it removes all local (to a procedure) symbols from the table, ready for the next procedure.

enter is the central routine in the symbol table handler. It is used both to enter new symbols into the table, and to look up existing routines. This is in fact done in the tokenizer. This means that the remainder of the compiler itself does not have to worry about identifier tokens as special: they simply contain a pointer to the symbol table slot they occupy. An implication of this is that undefined symbols are in fact entered into the table when they are first used. The first step in entry/look-up is to calculate a hash value for the string. This is simply the first character, shifted left six bits, along with the last character. For one-character names, these two are the same character. The resulting value is taken modulo the symbol table size to yield an index into the table at which to start the search for the symbol. Next comes a loop that starts at the given table position, and searches forward, either for an unused slot, or for one with the desired name. The forward search wraps around to the beginning of the table if it passes the end. If what is found is an empty slot, then the new symbol is added. Note that the name is copied into a dynamically-allocated region of memory. This is consistent with freeing the string when the symbol is removed. In either case, a pointer to the found table slot is returned. A slot kind of *sk_empty* tells the caller that the symbol has not been seen before.

The final routine in the file is *fixLocals*. This is used during code generation to fix up the offset from the frame pointer (stored in the *se_value* field) at which this local variable or parameter can be found. The fix is required since the offset amount is not known until all local variables have been declared and their required space added up. The extra offset could be remembered elsewhere and added in as required, but this method makes other code simpler.

The tokenizer

The tokenizer for the Toy compiler is in the file 'lex.d', shown in Listing 3. This portion of the compiler is more complex than the symbol table routines and maintains more internal states. In other compilers I have written, the current input token is always available for examination in a global variable. For the Toy compiler, however, I decided to work hard to keep things modular, so there are no global variables. The symbol table handler communicates solely through routines, for example. The interface to the tokenizer that I chose works similarly. Routine *nextToken* processes input characters to build up the next input token. The token is kept entirely within the tokenizer module ('lex.d'), however. The rest of the compiler calls *getToken* to return the current token. *getToken* can be called several times for a given token, and replaces the global variables I have used in the past.

The tokenizer maintains several distinct state systems. The simplest is simply a buffer to contain input text. This is composed of *SourceBuffer*, *SourcePos*, and *SourceMax*, which are,

respectively, the buffer for input characters, the position in the buffer of the next character to be processed, and the number of characters currently in the buffer. When the last character in the buffer has been processed, and more are needed, a call is made to the system interface portion of the compiler to read more characters.

A second state set is that which is used to build up identifiers and string constants. I prefer not to place limitations on the length of string contants, so a dynamically-sized buffer is used. Given the code to do that, there is no reason not to also use it to allow identifiers of arbitrary length. The variables involved here are *StringBuffer*, *StringBufferSize* and *StringBufferPos*, which are, respectively, a pointer to the dynamically allocated buffer, the size of the current buffer, and the position of the next character to be added to the buffer.

Also maintained is information that can be used in error messages to identify the location of the error. These are the number of the current input line, and the column of the current input character. In the Toy language, there are tokens that cannot be unambiguously identified without looking at the next input character. For example, if a < is seen, it is not known whether the token < is present, or if it is a <=. Looking ahead at a single next character is called one-character-lookahead. To handle this, the compiler has pairs of variables, containing the characters themselves, the line they came from, and the column they were in. This lookahead also adds some complexity to handling the end-of-file situation. The compiler should not get upset just because the lookahead is trying to read past the end of the input file. This is handled by the variable *EofState*, which has one of the following three values:

Eof_none: end of input has not been seen.
Eof_one: end of input seen for lookahead character.
Eof_two: end of input seen for current character.

Attempting to read another character when in the third state is an error. This typically happens with incomplete files or with comments that have not been closed.

The final set of state variables is used to describe the current input token. The variables are *CurrentToken*, *CurrentNumber*, *CurrentString* and *CurrentSymbol*. The first is always used, and is a member of the *TokenKind_t* enumeration from 'toy.g'. The second is used only for numeric constants. The third is used for string constants, and the fourth for identifiers. Recall from the discussion of the symbol table that all identifiers are inserted into the symbol table as soon as they are seen, so a pointer to their symbol table slot is sufficient to completely identify them.

The first procedures in the tokenizer that we should examine are the last two in the file. They are placed there because they call on the earlier routines, and I wanted to avoid unnecessary declarations. (Many people prefer a strict top-down ordering within a file; they will have to bear with my mostly-bottom-up arrangement.) *lexInit* is called on a per-file basis by the system interface routine. It is called *after* the *symbolInit* procedure, since it enters reserved words into the symbol table. The big declaration at the start of the function declares *RESERVED_WORDS* to be an array of *RESERVED_WORD_COUNT* structures of type *reservedWord_t*, the value of which is given directly at compile time. Note that this is a constant, not an initialized variable, as it would be in C. The difference usually doesn't matter, but is worth noting. This array contains the text of the various reserved words, the length of that text, and the token value that goes with it. *lexInit* goes through the array and calls *enter* to enter each one into the symbol table. They are given a kind of *sk_reservedWord*, and their token value as their *se_value* (subtracting *tk_eof* from their token value yields an integral value, which is compatible with the *se_value* field). Note that the reserved words are likely to hash to different places in the table, hence they are likely to be found without a search when they are looked up. This is desireable, since most reserved words occur more often than user identifiers.

Following the reserved word initialization, the string buffer is set up with a default-sized buffer (here 256 bytes). This will be sufficient for nearly all programs. The input source buffer is started off as being empty (*SourcePos* = *SourceMax*), so the first attempt to get a character will have to read some from the input file. The first call to *nextChar* will do this. At that point, due to the one-character lookahead, we will not yet have a value for the current character. The line and column counters are started to represent the position of the current character, which is obtained by the second call to *nextChar*. The final step of initialization is to construct the representation for the first token by calling *nextToken*. *lexTerm*, the tokenizer's terminnation routine, simply frees the current string buffer.

Now we go back to the beginning of the file. Routines *errorHere* and *errorHereThree* are used to generate error messages. They are called from the tokenizer itself, but are most used by the parsing routines to report syntactic and semantic errors. They are located in 'lex.d' so they can utilize the current position indicators in the messages they produce. They call on *errorPosition* and *printString*, which are in the system interface. The former prints the name of the input file and the passed line and column numbers in some reasonable format. The latter simply prints the passed string. *errorHereThree* is used for multipart messages, typically those containing a user program identifier. Note that *PreviousLine* and *PreviousColumn* are used as the position indicators. This is because the one-character-lookahead has read the next character, and *its* position is in *CurrentLine* and *CurrentColumn*.

Next is routine *nextChar*, which reads the next input character from the input stream. Recall that we have a buffer of characters from the input stream, and it is from there that *nextChar* should get the characters it needs. When that buffer-full is exhausted, the system interface should be called on to get another buffer-full. *nextChar*'s first action is to copy the lookahead character and its position into the current character and position. The position indicators for the next character are updated. If the character was a newline, the column goes to 0,

and the line is incremented. If the character was a tab, the column is moved up to the next tab stop. (Toy assumes the standard 8-character tab stops.) If *EofState* is *Eof_one*, we have already encountered the end of the input file for the lookahead character; we move up to *Eof_two*. Otherwise, we get the next character from the input buffer. If the input buffer is used up (*SourcePos* = *SourceMax*), we call *readSource* to read another buffer-full of characters. If none are available, then *EofState* is set to *Eof_one*. This type of activity could have been moved to the system interface file, but that would have required an additional subroutine call for each input character. Since the number of input characters read is the largest count of anything that a compiler has to deal with, it is best to keep tokenizing as efficient as possible. For simple compilers, the tokenizer can occupy up to 50 per cent of the total time.

Routine *addChar* implements the arbitrary-sized buffer used for string constants and identifiers. It is called to add a character to the buffer. If the buffer is full, it will allocate another buffer twice as big and copy over the current contents.

'Whitespace' is a term describing the spaces and empty lines in a file or document. Routine *whiteSpace* skips over this stuff in the input stream. It also skips over comments. Any amount of whitespace can occur between any two input tokens. This routine implements nested comments: comments that can be totally inside another comment. This is handy when experimenting with a program, since an entire section can be 'commented out' without worrying about whether it has any internal comments or not. This doesn't work with C programs (although **#if** can effectively do the same thing). Variable *level* is a count of how many comments we are currently nested in. It starts at zero for each region of whitespace and is incremented when /* is found and decremented when */ is found. The controlling expression of the **while** loop can be broken down as:

- Stop looping if we hit the end of the file.
- Loop past spaces, tabs and newlines.
- We are inside a comment if the current character is / and the next character is *.
- Loop while we are inside a comment.

Routine *nextToken* is the heart of the tokenizer. It uses the auxiliary routines we have seen and breaks up the input stream into tokens for the rest of the compiler. The nature of the token found is stored in the previously mentioned token state variables. The first step in this process is to skip any whitespace that separates the next token from the current one. Note that the convention is to skip the whitespace at the beginning of getting a token, rather than at the end. This allows the current-position variables to describe the last character of the current token. The tokenizer uses one-character-lookahead, but does not provide any token-level lookahead to the rest of the compiler. The language has been designed to not require any.

Again, since the tokenizer is a major time-consumer in a simple compiler, it should not spend too much time deciding what to do. The simplest way to handle this is used here: just do a big 'case' on the current input character to decide what kind of token we have. This builds a large case table, but it is worth the space in this instance. The local variable *tk* is used throughout this routine instead of the file level *CurrentToken*. This allows the use of a register variable for it. It is initialized to *tk_illegal* to allow the default case to be that of an unrecognized character.

The first case recognized is that of an identifier or a reserved word. They can begin with a capital or lower case letter, or with an underscore. The identifier is collected in the string buffer using *addChar*. The 'while' loop could be faster if table look-up were used to identify valid identifier characters. Note that digits have been added to the set of valid characters. A final **\e** (end-of-string) character is added to the buffer to mark the end of the identifier. Next, **enter** is called to look the name up in the symbol table. Recall that the routine will enter the symbol with kind *sk_empty* if it is not found. This is not handled here, but is passed along to the parser. The two cases here are that of the name being a reserved word, in which case the token kind is that of that reserved word, and that of the name being a regular (or unknown) identifier.

The next major case is that of a numeric constant. This is pretty straightforward: the value is built up and left in *CurrentNumber*. No check for overflow is made, and only decimal integers are supported. The corresponding code for Draco is much more complex, involving possible calls to the IEEE floating point library to convert a string to a floating point value.

The next case is the last major one. It is for string constants, which start with a " character. The code here is a bit more complex than might appear necessary. It does a number of things. First, it handles string breaks, that is, it will join together multiple string constants, separated by whitespace, into a single, large string constant. This allows long string constants on short input lines. Second, escaped characters within the string are supported. These are things like **\n** to represent a newline. The more complex cases are those like **\(23)**, representing a character with decimal value 23. These are used to produce things like the escape sequences used in the *shoot* program described in the previous article.

The outer 'while' loops through all the pieces of a string constant that are separated by string breaks. The next level loops through all of the characters in a given piece. It loops until it sees a closing " (or the end of the input file). Within the inner loop, \ is handled specially - when one is seen, the next character is read, and it is handled as an escaped character. Only newlines and tabs are handled directly here; among the other possiblities are **r** for a carriage return and **b** for a backspace. These can be obtained in Toy using the third escape alternative: a decimal number in parentheses. This form is used to maintain compatibility with Draco, which allows any constant expression to be within the parentheses. Again, no check for overflow is done on the decimal value.

Here we see our first two error messages in the compiler. They are complaints about not finding a) after the digits, or not

finding any digits after the (. The 'error recovery' here consists of doing absolutely nothing.

When a character has been read, whether escaped or not, it is added to the string buffer with *addChar*, and we move on to the next character. If it isn't a ", collecting the string piece continues. When a " is found, it is skipped via *nextChar*, and then whitespace is skipped as part of looking for another string piece. This violates the previously mentioned rule of skipping whitespace only at the start of a new token. As a consequence, an error message associated with a string constant will actually point just before the beginning of the *next* token. When the string is fully collected, a new buffer for it is allocated. It is pointed to by *CurrentString*, and its length is in *CurrentNumber*.

The remainder of the cases are simple ones that handle one or more operator or punctuation tokens. Note that the tilde (~) is illegal by itself, but can start the valid token ~=.

The remaining three routines represent the interface to the tokenizer that is available to the rest of the compiler. *getToken* returns any of the information pieces (*CurrentSymbol*, *CurrentString*, *CurrentNumber*) requested, along with the kind of the current token. In many situations, the compiler needs no more than the token kind, so *getSimpleToken* is an interface returning only that. *skipToken* is like *nextToken* in that it moves on to the next token. It, however, frees any storage (specifically, a string constant) associated with the token. It is used during error recovery when a particular token is being looked for.

Improvements

The symbol table routines are adequate for this compiler, but likely not for some other purposes. An interpreter, for instance, would probably need to be able to delete arbitrary definitions from the table. This would mess up the searching that *enter* does. Other compiler designs might benefit from multiple symbol tables. This would require dynamic allocation of the tables. Removing the limit on the number of symbols handled would also require that. The current hash function is not very good and would produce poor behaviour (lots of collisions) if the table gets too full. A better one could combine bits from the first two and last two characters of the symbol. This would add special cases for one-character symbols.

There isn't very much that can be done to a tokenizer to make it 'better'. A few changes could be done to speed it up, such as a table of characters types. Improvements would go along with changes to the language. For example, hexadecimal, octal and binary numeric constants are useful for some purposes. If floating point were added to Toy, the tokenizer would need to be able to convert strings of characters into floating point constants. Similarly, a character type would require character constants. Since they are a single occurrence of what is allowed inside string constants, it may then be desirable to extract that handling from the string processing and turn it into a separate routine. (Draco has a *nextStringChar* routine, for example.) More keywords could be added by extending the *TokenKind_t* enumeration and adding them to the table in *lexInit*. Similarly, additional operators and punctuation tokens would require additions to *TokenKind_t* and to the case statement in *nextToken*.

More ambitious changes would require major work elsewhere. Supporting full escaped characters as in Draco would require full constant expression evaluation elsewhere in the compiler. Adding a preprocessor as in C would insert another layer between these tokenizing routines and the system interface routines (or a separate program, but that would result in triple the amount of text file I/O).

Usage

How are these routines used? A full example can be found in the *tokenTest.d* file used to test the tokenizer. An outline of it goes something like this:

```
proc main()void:
   *SymbolEntry_t se;
   *char buffer;
   ulong number;
   TokenKind_t tk;

   symbolInit();
   lexInit();
   while
      tk := getToken(&se, &buffer, &number);
      tk ~= tk_eof
   do
      case tk
      incase tk_number:
         write('<', number, '>');
      incase tk_string:
         write('\"', buffer, '\"');
         memFree(buffer, number);
      incase tk_id:
         se*.se_kind := sk_undefined;
         write('<', se*.se_name, '>');
      ...
      esac;
      write(' ');
      nextToken();
   od;
   writeln();
   lexTerm();
   symbolTerm();
corp;
```

This program simply turns an input source file into a stream of tokens, which it prints; 'tokenTest.d' is an incomplete version of 'system.d' that provides only those services needed for this test.

Conclusion

This article has presented some information on the data structures used by the Toy compiler, and has described the symbol table and tokenizer modules. The symbol table routines serve

simply as a way to associate some information with a name. The tokenizer breaks a stream of input text into a stream of higher-level tokens, as requested by other parts of the compiler, and in accordance with the rules for tokens in the language. Neither job is overly complex, but both must be done right in order to make an acceptable compiler. Shown like this, they may seem obvious, but let me assure you that I didn't get them right the first time, even though I've done this a few times before!

Next month

The next article in this series will present the parser for the Toy compiler. There are three main portions to it: that which parses declarations, including procedure headers; that which parses expressions; and that which parses statements. It is in the parsing routines that the input program is 'understood' in some sense. It is also here that new language features can be added.

Listing 1: toy.g - the main include file

```
/*
 * toy.g: global types for the compiler.
 */

type

    /* type which enumerates all of the tokens the lexical scanner produces */

    TokenKind_t = enum {
        tk_eof,                 /* end of the source input reached */
        tk_illegal,             /* unknown characters in input */

        tk_number,              /* a numeric constant, e.g. 128 */
        tk_string,              /* a string constant */
        tk_id,                  /* an identifier */

        tk_proc,                /* the reserved word 'proc' */
        tk_corp,
        tk_void,
        tk_int,
        tk_bool,
        tk_true,
        tk_false,

        tk_if,
        tk_then,
        tk_elif,
        tk_else,
        tk_fi,
        tk_while,
        tk_do,
        tk_od,
        tk_readln,
        tk_write,

        tk_assign,              /* the assignment operator, ':=' */
        tk_leftParen,
        tk_rightParen,
        tk_semicolon,
        tk_colon,
        tk_comma,
        tk_plus,
        tk_minus,
        tk_slash,
        tk_star,
        tk_equal,
        tk_notEqual,
        tk_less,
        tk_lessEqual,
        tk_greater,
        tk_greaterEqual
    },

    /* type which describes the toy-type of an expression or call */

    ResultType_t = enum {
        rt_void,
        rt_bool,
        rt_int,
        rt_error                /* unknown type */
    },

    /* type which enumerates the various kinds of symbol-table entries */

    SymbolKind_t = enum {
        sk_empty,               /* slot is empty in table */
        sk_reservedWord,        /* slot used for a reserved word */
        sk_undefined,           /* an undefined symbol */
        sk_globalVariable,
        sk_procedure,
        sk_parameter,
        sk_localVariable
    },

    /* type which describes each symbol-table entry */

    SymbolEntry_t = struct {
        *char se_name;          /* points to text of name of symbol */
        ulong se_length;        /* length of name, including '\e' */
        SymbolKind_t se_kind;   /* what kind of symbol it is */
        ResultType_t se_type;   /* what it's type is */
        uint se_parCount;       /* count of parameters (for procs) */
        int se_value;           /* value - used by code generation */
    },

    /* type which enumerates the various relocation targets */

    RelocCode_t = enum {
        rc_none,                /* a marker only */
        rc_globalVariable,      /* reference to a global variable */
        rc_main,                /* reference to 'main' by run-time system */
        rc_printString,         /* ref to run-time routine 'printString' */
        rc_printInt,
        rc_readLine,
        rc_readInt
    };
```

Listing 2: symbol.d - the symbol table code

```
#drinc:util.g
#toy.g
#externs.g

/*
 * symbol.d: routines for maintaining symbol tables.
 */

ulong SYMBOL_TABLE_SIZE = 1023;

ulong EntryCount;
[SYMBOL_TABLE_SIZE]SymbolEntry_t SymbolTable;

/*
 * symbolInit - initialize the symbol table.
 */

proc symbolInit()void:
    register *SymbolEntry_t se;
    register uint i;

    se := &SymbolTable[0];
    for i from 0 upto SYMBOL_TABLE_SIZE - 1 do
        se*.se_kind := sk_empty;
        se := se + sizeof(SymbolEntry_t);
    od;
    EntryCount := 0;
corp;

/*
 * symbolTerm - clean up the symbol table stuff.
 */

proc symbolTerm()void:
    register *SymbolEntry_t se;
    register uint i;
```

```
    se := &SymbolTable[0];
    for i from 0 upto SYMBOL_TABLE_SIZE - 1 do
        if se*.se_kind ~= sk_empty then
            se*.se_kind := sk_empty;
            memFree(se*.se_name, se*.se_length * sizeof(char));
        fi;
        se := se + sizeof(SymbolEntry_t);
    od;
corp;

/*
 * symbolPurge - purge all local symbols from the symbol table.
 */

proc symbolPurge()void:
    register *SymbolEntry_t se;
    register uint i;

    se := &SymbolTable[0];
    for i from 0 upto SYMBOL_TABLE_SIZE - 1 do
        if se*.se_kind = sk_parameter or se*.se_kind = sk_localVariable then
            se*.se_kind := sk_empty;
            memFree(se*.se_name, se*.se_length * sizeof(char));
            EntryCount := EntryCount - 1;
        fi;
        se := se + sizeof(SymbolEntry_t);
    od;
corp;

/*
 * enter - enter/lookup a symbol in the table. If the symbol is not found,
 *      we enter it, but leave its entry type as sk_empty. When we enter
 *      it, we copy the string over.
 */

proc enter(register *char name; register ulong length)*SymbolEntry_t:
    register *SymbolEntry_t se;

    se := &SymbolTable[0] +
            ((make(name* - '\e', uint) << 6) |
              make((name + (length - 2) * sizeof(char))* - '\e', uint)) %
            SYMBOL_TABLE_SIZE * sizeof(SymbolEntry_t);
    while se*.se_kind ~= sk_empty and
        (se*.se_length ~= length or not CharsEqual(se*.se_name, name))
    do
        se :=
            if se = &SymbolTable[0] then
                se + (SYMBOL_TABLE_SIZE - 1) * sizeof(SymbolEntry_t)
            else
                se - sizeof(SymbolEntry_t)
            fi
    od;
    if se*.se_kind = sk_empty then
        if EntryCount = SYMBOL_TABLE_SIZE - 1 then
            abort("symbol table overflow");
        fi;
        EntryCount := EntryCount + 1;
        se*.se_name := memAlloc(length * sizeof(char));
        BlockCopy(se*.se_name, name, length * sizeof(char));
        se*.se_length := length;
    fi;
    se
corp;

/*
 * fixLocals - add an offset to all local symbols.
 */

proc fixLocals(int extraOffset)void:
    register *SymbolEntry_t se;
    register uint i;

    se := &SymbolTable[0];
    for i from 0 upto SYMBOL_TABLE_SIZE - 1 do
        if se*.se_kind = sk_parameter or se*.se_kind = sk_localVariable then
            se*.se_value := se*.se_value + extraOffset;
        fi;
        se := se + sizeof(SymbolEntry_t);
    od;
corp;
```

Listing 3: lex.d - the Toy tokenizer

```
#drinc:util.g
#toy.g
#externs.g

/*
 * lex.d: lexical scanner - break input stream into tokens.
 */

uint
    DEFAULT_STRING_BUFFER_SIZE = 256,
    SOURCE_BUFFER_SIZE = 512;

/* a buffer to hold an input string constant or identifier */

*char StringBuffer;
ulong StringBufferSize, StringBufferPos;

/* a buffer to hold input text from the source file */

[SOURCE_BUFFER_SIZE]char SourceBuffer;
uint SourcePos, SourceMax;

/* our current state during the lexical scan */

uint CurrentLine, PreviousLine, CurrentColumn, PreviousColumn;
char CurrentChar, NextChar;
enum {Eof_none, Eof_one, Eof_two} EofState;
*SymbolEntry_t CurrentSymbol;
*char CurrentString;
ulong CurrentNumber;
TokenKind_t CurrentToken;


/*
 * errorHere - signal an error occurring at the current input position.
 */

proc errorHere(*char message)void:

    errorPosition(PreviousLine, PreviousColumn);
    printString(message);
    printString("\n");
corp;

/*
 * errorHereThree - a three-part error message.
 */

proc errorHereThree(*char m1, m2, m3)void:

    errorPosition(PreviousLine, PreviousColumn);
    printString(m1);
    printString(m2);
    printString(m3);
    printString("\n");
corp;

/*
 * nextChar - get the next input character.
 */

proc nextChar()void:

    CurrentChar := NextChar;
    PreviousLine := CurrentLine;
    PreviousColumn := CurrentColumn;
    if CurrentChar = '\n' then
        CurrentLine := CurrentLine + 1;
        CurrentColumn := 0;
    elif CurrentChar = '\t' then
        CurrentColumn := CurrentColumn / 8 * 8 + 8;
    else
        CurrentColumn := CurrentColumn + 1;
    fi;
    if EofState = Eof_one then
        EofState := Eof_two;
        NextChar := ' ';
```

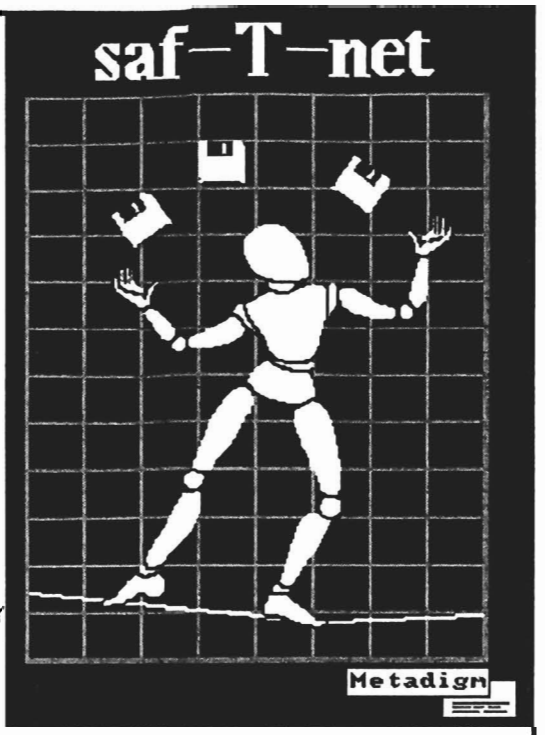
```
    else
        if SourcePos = SourceMax then
            SourceMax := readSource(&SourceBuffer[0], SOURCE_BUFFER_SIZE);
            if SourceMax = 0 then
                SourcePos := 0;
                EofState := Eof_one;
                NextChar := ' ';
            else
                NextChar := SourceBuffer[0];
                SourcePos := 1;
            fi;
        else
            NextChar := SourceBuffer[SourcePos];
            SourcePos := SourcePos + 1;
        fi;
    fi;
corp;

/*
 * addChar - add a character to the current string buffer.
 */

proc addChar(char ch)void:
    *char newBuffer;
    ulong newSize;

    if StringBufferPos = StringBufferSize then
        newSize := StringBufferSize * 2;
        newBuffer := memAlloc(newSize * sizeof(char));
        BlockCopy(newBuffer, StringBuffer, StringBufferSize * sizeof(char));
        memFree(StringBuffer, StringBufferSize * sizeof(char));
        StringBufferSize := newSize;
        StringBuffer := newBuffer;
    fi;
    (StringBuffer + StringBufferPos * sizeof(char))* := ch;
    StringBufferPos := StringBufferPos + sizeof(char);
corp;

/*
 * whiteSpace - skip past spaces, tabs, newlines, comments.
 */

proc whiteSpace()void:
    register uint level;

    level := 0;
    while EofState ~= Eof_two and
        (CurrentChar = ' ' or CurrentChar = '\t' or CurrentChar = '\n' or
         CurrentChar = '/' and NextChar = '*' or
         level ~= 0)
    do
        if CurrentChar = '/' and NextChar = '*' then
            level := level + 1;
            nextChar();
        elif CurrentChar = '*' and NextChar = '/' then
            level := level - 1;
            nextChar();
        fi;
        nextChar();
    od;
corp;

/*
 * nextToken - get the next input token.
 */

proc nextToken()void:
    register TokenKind_t tk;
    register int n;

    whiteSpace();

    if EofState = Eof_two then
        tk := tk_eof;
    else
        tk := tk_illegal;
        case CurrentChar
        incase 'a'..'z':
        incase 'A'..'Z':
        incase '_':
            StringBufferPos := 0;
            while (CurrentChar >= 'a' and CurrentChar <= 'z' or
                CurrentChar >= 'A' and CurrentChar <= 'Z' or
                CurrentChar >= '0' and CurrentChar <= '9' or
                CurrentChar = '_') and EofState ~= Eof_two
            do
                addChar(CurrentChar);
                nextChar();
            od;
            addChar('\e');
            CurrentSymbol := enter(StringBuffer, StringBufferPos);
            if CurrentSymbol*.se_kind = sk_reservedWord then
                tk := CurrentSymbol*.se_value + tk_eof;
            else
                tk := tk_id;
            fi;
        incase '0'..'9':
            tk := tk_number;
            CurrentNumber := 0;
            while CurrentChar >= '0' and CurrentChar <= '9' and
                EofState ~= Eof_two
            do
                CurrentNumber := CurrentNumber * 10 + (CurrentChar - '0');
                nextChar();
            od;
        incase '\"':
            tk := tk_string;
            StringBufferPos := 0;
            while
                nextChar();
                while CurrentChar ~= '\"' and EofState ~= Eof_two do
                    if CurrentChar = '\\' then
                        nextChar();
                        if CurrentChar = 'n' then
                            CurrentChar := '\n';
                        elif CurrentChar = 't' then
                            CurrentChar := '\t';
                        elif CurrentChar = '(' then
                            nextChar();
                            if CurrentChar >= '0' and CurrentChar <= '9' then
                                n := 0;
                                while CurrentChar >= '0' and CurrentChar <= '9'
                                do
                                    n := n * 10 + (CurrentChar - '0');
                                    nextChar();
                                od;
                                if CurrentChar = ')' then
                                    CurrentChar := n + '\e';
                                else
                                    errorHere(
                                        "missing ')' in escaped character");
                                fi;
                            else
                                errorHere(
                                    "expecting digits for escaped character");
                            fi;
                        fi;
                    fi;
                    addChar(CurrentChar);
                    nextChar();
                od;
                nextChar();
                whiteSpace();
                EofState ~= Eof_two and CurrentChar = '\"'
            do
            od;
            addChar('\e');
            CurrentString := memAlloc(StringBufferPos * sizeof(char));
            BlockCopy(CurrentString, StringBuffer,
                    StringBufferPos * sizeof(char));
            CurrentNumber := StringBufferPos;
        incase '=':
            tk := tk_equal;
            nextChar();
        incase '~':
            if NextChar = '=' then
                tk := tk_notEqual;
                nextChar();
                nextChar();
            else
                nextChar();
            fi;
```

```
        incase '<':
            if NextChar = '=' then
                tk := tk_lessEqual;
                nextChar();
            else
                tk := tk_less;
            fi;
            nextChar();
        incase '>':
            if NextChar = '=' then
                tk := tk_greaterEqual;
                nextChar();
            else
                tk := tk_greater;
            fi;
            nextChar();
        incase ':':
            if NextChar = '=' then
                tk := tk_assign;
                nextChar();
                nextChar();
            else
                tk := tk_colon;
                nextChar();
            fi;
        incase '+':
            tk := tk_plus;
            nextChar();
        incase '-':
            tk := tk_minus;
            nextChar();
        incase '*':
            tk := tk_star;
            nextChar();
        incase '/':
            tk := tk_slash;
            nextChar();
        incase '(':
            tk := tk_leftParen;
            nextChar();
        incase ')':
            tk := tk_rightParen;
            nextChar();
        incase ',':
            tk := tk_comma;
            nextChar();
        incase ';':
            tk := tk_semicolon;
            nextChar();
        default:
            nextChar();
        esac;
    fi;
    CurrentToken := tk;
corp;

/*
 * getToken - return the current token.
 */

proc getToken(**SymbolEntry_t pSymbol; **char pBuffer;
              *ulong pNumber)TokenKind_t:

    if pSymbol ~= nil then
        pSymbol* := CurrentSymbol;
    fi;
    if pBuffer ~= nil then
        pBuffer* := CurrentString;
    fi;
    if pNumber ~= nil then
        pNumber* := CurrentNumber;
    fi;
    CurrentToken
corp;

/*
 * getSimpleToken - get a token, ignoring anything except the token kind.
 */

proc getSimpleToken()TokenKind_t:

    CurrentToken
corp;

/*
 * skipToken - totally skip over the current token.
 */

proc skipToken()void:

    if CurrentToken = tk_string then
        memFree(CurrentString, CurrentNumber);
    fi;
    nextToken();
corp;

/*
 * lexInit - initialize for lexical scanning.
 */

proc lexInit()void:
    uint
        MAX_RESERVED_WORD_LENGTH = 6,
        RESERVED_WORD_COUNT = 17;
    type reservedWord_t = struct {
        [MAX_RESERVED_WORD_LENGTH + 1]char rw_name;
        ulong rw_length;
        TokenKind_t rw_token;
    };
    [RESERVED_WORD_COUNT] reservedWord_t RESERVED_WORDS = (
        ("proc"  , 4 + 1, tk_proc  ),
        ("corp"  , 4 + 1, tk_corp  ),
        ("void"  , 4 + 1, tk_void  ),
        ("int"   , 3 + 1, tk_int   ),
        ("bool"  , 4 + 1, tk_bool  ),
        ("true"  , 4 + 1, tk_true  ),
        ("false" , 5 + 1, tk_false ),
        ("if"    , 2 + 1, tk_if    ),
        ("then"  , 4 + 1, tk_then  ),
        ("elif"  , 4 + 1, tk_elif  ),
        ("else"  , 4 + 1, tk_else  ),
        ("fi"    , 2 + 1, tk_fi    ),
        ("while" , 5 + 1, tk_while ),
        ("do"    , 2 + 1, tk_do    ),
        ("od"    , 2 + 1, tk_od    ),
        ("readln", 6 + 1, tk_readln),
        ("write" , 5 + 1, tk_write )
    );
    register *reservedWord_t rw;
    register *SymbolEntry_t se;
    register uint i;

    rw := &RESERVED_WORDS[0];
    for i from 0 upto RESERVED_WORD_COUNT - 1 do
        se := enter(&rw*.rw_name[0], rw*.rw_length);
        se*.se_kind := sk_reservedWord;
        se*.se_value := rw*.rw_token - tk_eof;
        rw := rw + sizeof(reservedWord_t);
    od;
    StringBuffer := memAlloc(DEFAULT_STRING_BUFFER_SIZE);
    StringBufferSize := DEFAULT_STRING_BUFFER_SIZE;
    SourcePos := 0;
    SourceMax := 0;
    EofState := Eof_none;
    nextChar();
    CurrentLine := 1;
    CurrentColumn := 1;
    nextChar();
    nextToken();
corp;

/*
 * lexTerm - terminate the lexical scanner stuff.
 */

proc lexTerm()void:

    if StringBuffer ~= nil then
        memFree(StringBuffer, StringBufferSize * sizeof(char));
    fi;
corp;
```

NEW Release

ver 1.10

- Full Intuition based Interface
- Selective Restore
- Amiga wildcards
- Multiple indexing for greater **safety**
- Over 907K per floppy



Hard Disk Backup

for your Amiga

available at your dealers **now**

or from (714) 955-2555

Metadigm, Inc. 19762 MacArthur Blvd. Suite 300; Irvine CA 92715

Free Spirit Software Inc.

AMI..ALIGNMENT SYSTEM

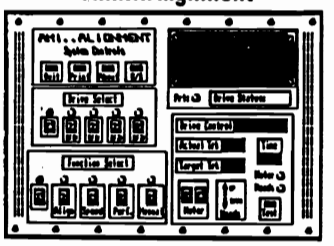
Precision drive evaluation and head alignment system for the Amiga series of computers. The Ami..Alignment System evaluates the mechanical condition of internal and external 3.5" disk drives and provides information for the correction of non-standard conditions.

From a control panel menu, head alignment, drive speed and read/write performance may be tested. Instantaneous readouts are displayed on screen or may be output to printer.

Includes program disk, calibration disk and operations manual. Instructions and diagrams for dissassembly and adjustment of Commodore and third party drives are included.

Diagnose and treat the "flaky floppy" syndrome.

Ami...Alignment



Treat your drives with respect!

Suggested retail price $49.95

See your local dealer or contact:
Free Spirit Software
58 Noble St.
Kutztown, PA 19530
215-683-5609
FAX #215-683-8567

GAME PROGRAMS WANTED!

Attention Developers and Freelance Programmers!

Virtual Reality Laboratories, Inc. is looking for a few sharp new entertainment game and educational game programs for the Amiga. To be accepted, these programs must be highly graphic with excellent sound capability. For a rapid review of your work send for guidelines to VRLI, Box 609, Atwood, CA 92601.

Do not submit until you have received and read our guidelines, but please do send for the guidelines TODAY!

. . . if you have at least 3 years of assembly language programming experience, preferably one year of 68000 or have an NTSC/PAL Video Hardware background. Join the team that makes no compromises and help us change the world. Key positions available at the fastest growing Amiga company. Hackers call Stephanie at 913-354-1146 or write to Stephanie at 115 West Crane Street, Topeka KS 66603.

Copying Disks

The use of trackdisk.device

by Bob Rakosky
Copyright © 1989 Bob Rakosky

Bob Rakosky is a data processing consultant for AGS Information Services, Inc. He is the author of the hard-disk back-up utility ***saf-T-net*** *(published by Metadigm), and of the Amiga version of the game* ***Empire*** *(published by InterStel).*

The Amiga floppy disks can be accessed via a number of levels. The standard way that we read from and write to disks is by means of the *file system*, which manages the space on the disks, and keeps track of logical groupings of data called files. It is through the file system that we can get a directory listing of the available files, find out how much space is currently unallocated on the diskette, read the data that is contained in files, and create new files. The file system is device-independent. It sees the data on any file system device as a collection of blocks, each containing 512 bytes. All it needs to know about a device is the number of 512-byte blocks of data that are available. It reads and writes data in terms of the relative block number, starting at relative block 0 and continuing to relative block n-1, where n is the number of blocks that the device can hold.

Since my desire is to make a copy of a diskette, I could accomplish this by using the file system, assuming that I already had a blank, formatted disk onto which I would write all of the files found on the source disk. But what if I didn't have a formatted diskette? What if the disk that I'm copying wasn't created by the AmigaDOS file system, and contains data not grouped into 'files'? Even if the disk is a standard file system disk, copying it file-by-file can be pretty slow, especially if the disk contains lots of small files. Copying at the device level, rather than the file system level, can be much faster, as long as we are copying from and to identical devices. This is the level at which I will be accessing the floppy disks.

There is yet another level by which we could access the floppy drives: by going directly to the hardware. Since my intention is to only access standard Amiga format diskettes, this approach would be overkill for our needs and won't be addressed in this article.

Data organization

The trackdisk.device driver views the data physically stored on the medium as a series of tracks of encoded data. Each track on the Amiga diskette holds 11 sectors of data (plus a little extra information not used by the file system), and each sector contains 512 bytes of data, after it is decoded. (There need not be an exact correspondence between the 512-byte block size used by the file system and the 512-byte sector size of the trackdisk, but it does simplify things.) The actual layout of the data within the track is unimportant for our purposes, as those details are handled for us by the device driver. We don't have to know how the data is encoded, nor do we have to know how to decode the control information contained on the physical media. A given device is configured to contain a fixed number of tracks per surface, and will contain a fixed number of surfaces. For the standard Amiga 3.5" floppy drives, each surface contains 80 tracks, and the diskette contains two usable surfaces. For the standard Amiga 5.25" drive, the geometry is two surfaces of 40 tracks each.

Input and output requests are made to the trackdisk device in terms of the *relative byte address*, starting from 0 and continuing up to the capacity of the drive (880K for the 3.5" drives). Requests must be made only to addresses that fall on a sector boundary (the byte address must be a multiple of 512), and the length of data read or written (also expressed in bytes) must be *full sector* amounts (also multiples of 512). The device driver will only read or write data in terms of full tracks, however, and so must maintain an internal buffer to hold the entire track being accessed. Since we will be reading and writing entire disks, it makes more sense to perform all of our IO operations in full track units, since that is how the data will be manipulated internally by the driver.

Using device drivers

The Amiga supports its peripheral hardware devices via software modules called *device drivers*. The driver for the floppy disks is contained in the KickStart code, and it is accessed in the same way that all Amiga devices are accessed. Access to the device is obtained via the OpenDevice() Exec library function, to which you specify the *device name* and the *unit number*. After this, input and/or output commands are sent to the device using the DoIO() or SendIO() functions, passing an IORequest block structure. These two functions are similar, except that the DoIO() function call will not return until the

actual input/output action has completed, whereas SendIO() will schedule the IO action, but will possibly return before the operation has actually completed. In the latter case, your program is free to perform other processing while the actual IO is being performed. The communication between your task and the (separate) task that actually performs the IO processing is accomplished by Exec's message-passing facilities (the first part of the IORequest structure is an Exec Message structure). In the case of asynchronous IO, this message is ReplyMsg()'ed by the device-driver task to be received by your task at the mn_ReplyPort (MsgPort) contained in the Message. This can then be Wait()'ed for, or tested for via GetMsg().

A full discussion of device IO on the Amiga would take much more space than this article will allow. A full discussion, on a general level, takes up a full chapter in the ROM Kernal Manuals. We will limit our discussion to issues directly related to the accessing of floppy disks, as illustrated in the program source listing accompanying this article.

IO via trackdisk.device

The trackdisk.device supports three groupings of command requests: standard device commands, trackdisk-specific commands, and 'extended' trackdisk-specific commands. The accompanying program, *copydisk.c*, utilizes commands that fall into the first two of these categories. The commands we will be discussing and using are:

Standard Commands

CMD_READ	Reads one or more sectors
CMD_WRITE	Writes one or more sectors

Trackdisk-specific Commands

TD_MOTOR	Turns the motor on or off
TD_FORMAT	Initializes one or more tracks
TD_GETNUMTRACKS	Inquire as to the total number of tracks supported by the device.

As stated above, device-level IO is performed using a structure called an IORequest. The trackdisk.device uses an extended form of an IORequest called an IOStdReq structure. Both of these structures are defined in the standard include file *<exec/io.h>*. In addition, the 'extended' trackdisk-specific commands use an extended form of the IOStdReq structure. This extended IORequest structure, called an IOExtTD structure and defined in the header file *<devices/trackdisk.h>*, is not used in the accompanying program, as the extended commands are not needed nor used in the program. The ensuing discussion will refer to the IORequest structure, without differentiating between these various extended versions.

Putting it all together: *Copydisk* the program

The accompanying program, *Copydisk.c*, was written to illustrate some of the principles involved with reading and writing floppy disks at the device driver level. It is a simple disk copier program, with a couple of advantages over the standard DiskCopy program that's distributed with the Amiga Workbench. The program is designed to take advantage of the available memory in the system to allow you to make copies with a single disk drive with as few disk swaps as possible. Also, if your system has enough memory to hold the entire source disk's contents in memory, you can make multiple copies of the same source disk, without having to reread the source for every copy made.

The program is designed to be run from the CLI only. The command line to invoke it is of the form:

```
copydisk src-device dest-device [MULTI] [NOVERIFY]
```

where both the source and destination devices are valid disk drive identifiers such as "DF0:" or "DF2:". The MULTI option, if specified, requests that the program not terminate after completing the first copy, but to continue prompting the user to allow multiple copies of the source disk to be made. This mode is terminated when the user enters a CTRL-C from the keyboard. The NOVERIFY option, if requested, will cause the data to be written to the destination disk more quickly, but provides a lot less error checking. It should be used with caution, as it is quite possible for a write error to go undetected in this mode.

I decided to use some functions in *arp.library*, the freely-distributable library of support functions created/released by Charlie Heath, Scott Ballantyne, and others. The program uses the ARP start-up code as distributed with the ARP Version 1.1 release (the current release available at the time I am writing this), instead of the standard start-up code distributed with the C compiler package. The advantage this provides is that the arp.library is automatically opened by the start-up package, and the command line is parsed automatically, via a call to the ARP Gads() function, prior to control being passed to the main() function. Consequently, the argument processing in the main() function can safely assume that the argument pointers in the argv[] array are in the order laid out in the command-line template as specified in the *CLI_Template variable.

The primary advantage provided to this program by the use of ARP is the reduction of the amount of code that I had to write. This savings was particularly important to me, as I know that the editors of this magazine have limitations on the size of a program listing that they can print. The ARP startup allows me to have sophisticated command-line parsing and prompting, without having to add the code myself. I also made use of the ARP memory-tracking (the ArpAllocMem() function), which allowed me to allocate the linked list of cache buffers, and bypass freeing them, since they are freed automatically during the program termination (thanks again to the closing of arp.library automatically by the _exit code contained in the ARP start-up. Further savings were realized by using the ARP function to perform the DOS packet-level IO required to stop the file system from using a disk drive (refer to the function inhibit_drives() in the accompanying program listing).

However, I did run into one potential problem, using the ARP start-up code. In the function valid_devs(), I attempt to walk the linked list of device nodes in the system, in order to validate the devices specified in the command line. The way to find the beginning of this device list is via the RootNode structure, that is pointed to (via a BPTR) from within the DOS Library structure, whose address is given by the standard DosBase pointer obtained via opening "dos.library" in the start-up code. However, when using the ARP start-up, DosBase really contains a pointer to the ARP library base (which contains a semi-clone of the DOS library). Unfortunately, this 'clone' is not an exact copy, and some fields contained in the DosLibrary structure are not set in the copy. In the 1.1 version of ARP library that I used, the BPTR to the RootNode was not set. (Note: I have been informed that with the 1.3 version of the ARP library, which is in Beta testing as I write this, the RootNode pointer *is* available from this pseudo-DosBase pointer). Luckily, the real DosBase pointer is saved in the ArpBase Library structure, so I was able to use it to get the address of the RootNode. From there, I was able to find the system device node list and perform the validations needed.

Examining the disk IO routines

The first step in accessing the floppy disk is to issue an OpenDevice() call for the device. This will initialize an IORequest structure with the needed values in order to perform IO requests. The values set in this IORequest structure will be needed to send IO requests to the driver. They could be copied to another allocated IORequest structure if needed (by copying the entire structure that is initialized by OpenDevice()). In our case, we will just initialize a single IORequest for the input device and use it for all of the input. The same process is followed for the output device.

We now need to determine the total number of tracks supported by the source device. This will tell us the maximum number of cache buffers to allocate, since we wish to attempt to cache as much of the source disk as we can fit into memory (to minimize the number of disk swaps needed, if copying from and to the same drive, and to enable us to perform multiple writes of the disk from the single read copy of the source if possible). We also need to verify that the output device utilizes the same number of tracks, as our logic depends on it. The trackdisk command TD_GETNUMTRACKS will return the number of tracks available on the disk unit in question. This value is returned in the field io_Actual in the IORequest structure.

After we have successfully gained access to the device driver, we need to tell the AmigaDOS file system to stay away from our disks. This is because it would be dangerous for the file system to have access to the device while we have partially written to it, since we will not be maintaining the logical integrity of the file data between the time that we start writing to the destination disk and when we complete the copy operation. This 'keep your hands off' request is handled by the routine inhibit_drives(), which utilizes DOS packet-level IO to communicate with the file system. Since this article is about accessing the device driver rather than accessing the file system handler, I'm not going to go into extensive detail about this. Suffice it to say that, by sending an ACTION_INHIBIT packet to the handler with an argument value of TRUE, the file system will consider the disk to be the standard "Not a DOS Disk", and will not try to access it. The disk Icon, if Workbench is loaded, will show the "DFn:BUSY" legend.

After allocating the buffers we will need to cache the data read, we still need to allocate at least one buffer into which we will read the data and from which we will write the data. This is because the data area passed to the trackdisk.device for input and output operations *must be in Chip RAM*. This is a requirement of the trackdisk device, primarily because the device driver uses the Amiga's blitter to perform the encoding and decoding of the data (the data that is physically on the floppy disk media is not stored as 8 bits of data per character as it it in memory, but is encoded using a technique called MFM, which helps to ensure that the series of bits is interpreted correctly). We aren't allocating our cache buffers specifically in Chip memory, because it isn't possible to fit an entire floppy disk into the available Chip area (until the very fat one is available).

This leads us to another design decision that I had to make in writing this program. I decided to use *synchronous* IO, which means that my IO requests will not return to my program until they have completed. This is accomplished by using the DoIO() function call. I could have chosen to perform *asynchronous* IO, using the SendIO() function, in which case my program would get control after the request was accepted, but before it completed. That would enable me to, for example, read one track while I was writing another (assuming that I was reading from and writing to different disk drives. I could also issue a second read request while the first one was processing, but this would not result in much faster operation. While the first example (simultaneous reading and writing) would result in greater throughput, I chose not to implement it, primarily because it would add more complexity to the program, and I wanted to attempt to keep the example code simple and straightforward. (I also needed to try to keep the size down, so the publishers wouldn't hang me.)

As noted above, the trackdisk supports reads and writes in any multiple of the sector size. However, since the data is transferred to and from the device in full tracks only, it is more efficient for us to design our input and output accordingly and read/write full tracks at a time. The reading of the data is pretty simple: we set up the request using the standard IO command CMD_READ, specify the beginning offset (in bytes) as the current track number (relative to zero) times the track size, and request to read a full-track length. This is read into our Chip memory buffer and copied from there to our cache buffer.

Writing the track of data could have been done in an identical manner, using the standard IO command CMD_WRITE. However, the accompanying program does not use this technique. The *copydisk* program uses the trackdisk-specific command TD_FORMAT instead. This requires a bit of explanation.

The way the trackdisk device handles CMD_WRITEs is a little different than one might expect. In order to write any data to a track on the disk, it first reads in the entire track from the disk, then replaces within its internal buffer the sector(s) that the program is actually writing. It then will write out the full track from its buffer. In our case, it would read in the entire track, replace the entire track in its buffer with the data we're writing (since we're writing the full track), and then write the data out. This works as long as the disk we are writing to is already formatted; otherwise, the initial read will fail, and the write won't be performed. However, it is considerably slower, since it requires the extra read of the full track before the write takes place.

There is an alternative, however, since we are performing full-track writes only. The trackdisk TD_FORMAT command will simply write out a track's worth of data, overlaying whatever happened to be on that track of the disk previously. This will work even for an unformatted disk, since this is the command that allows the disk to be formatted. The only restriction is that the operation must be performed in full-track increments. The negative side to doing the writes via the TD_FORMAT command is that the command will appear to succeed even if there is a problem with the disk (such as a bad diskette). This will only be detected with a subsequent read of the track. This is why the program issues a CMD_READ request after the TD_FORMAT, unless the user decides to gamble by specifying the NOVERIFY command-line option. With the verify enabled, the operation is not any faster than it would have been with the CMD_WRITE, but we get a better test of the data on the output medium (since we read *after* we write). This technique will also work on an unformatted diskette, whereas the CMD_WRITE technique would not.

One other command deserves some mention here: TD_MOTOR. As previously noted, this command is used to turn the floppy drive's motor on or off. The observant reader will notice that the program listing never contains any IO requests to turn the drive motor on. This is because the trackdisk device will automatically turn the drive's motor on when needed. However, it does not turn the motor off, unless explicitly requested by the application program. Consequently, we did have to include requests to turn the motor off at appropriate times, such as when we wish the user to replace the disk in the drive with another. It's not a particularly good idea to leave the motor on (and the accompanying LED light), and ask the user to swap disks: it leaves them a bit on the confused side. We don't automatically turn the motor off after each read or write, though, since that would slow things down. It does take a significant amount of time for the motor to get up to speed, and we only want to incur this overhead when it is really necessary.

Finally, it should be noted that, by doing a track-by-track copy of a disk, the resulting disk is an *exact duplicate* of the original. The Amiga file system keeps track of the diskettes it sees, and gets very confused if it sees two identical diskettes; generally, your machine will either 'guru' or lock up. The way to avoid this is to ensure that the file system sees some difference in the two disks. We do this by arranging for the disk creation date (contained in the Root Block of the disk) to be different for the two disks. I refer the reader to the functions write_track() and touch_root_blk() in the example program.

Source code with a dual personality

Copydisk was developed using Lattice C, version 5.02, in the standard 32-bit integer mode. However, it was written to be compiled as-is using the current version (3.6a) of the Manx compiler in its standard 16-bit mode. As a result, certain coding constructs may appear to be a bit more complicated than those to which you are accustomed. For example, I definitely wanted to make use of the function-prototyping capability of the Lattice compiler, which is currently unsupported by Manx's Aztec C. Consequently, I use the **__ARGS()** macro in declaring the prototypes. This macro is defined, for the Lattice package, in the header file *<stdio.h>*, with:

```
#define __ARGS(a) a
```

and is defined (by me) for the Manx compiler (within an **#ifdef** AZTEC_C block) with:

```
#define __ARGS(a) ()
```

This allows the function prototype statement in the form:

```
int read_track __ARGS((struct cache *));
```

to be expanded, for Lattice, as:

```
int read_track (struct cache *);
```

and, for Manx, as:

```
int read_track ();
```

which are the appropriate forms under the respective compilers for a forward reference/definition of a function.

The difference in integer size in the default modes of both compilers caused a few incompatibilities. This resulted in some extra casts that might appear unneeded in one implementation or the other, but allows the source code to work correctly with both systems.

In conclusion...

Copydisk.c illustrates some techniques used in accessing the floppy disk drives at the device driver level. It is also a useful disk-copier program, but that is a secondary benefit. Because of this, there are some shortcomings to the code as presented. There are also some opportunities (left to the reader as a homework exercise) for substantially improving the program and its capabilities. Better error-handling logic is needed. Desirable enhancements include the adding of asynchronous IO logic (which is not quite as trivial as it may at first seem). Also on my wish-list is a real Intuition-style interface so that the program is not restricted to CLI use. But it's a start!

Listing: Copydisk.c

```
/*****************************************************************************/
/*                                                                           */
/*                              COPYDISK V1.0                                */
/*                                                                           */
/*                     Copyright (c) 1989 Bob Rakosky                        */
/*                                                                           */
/*  Usage:                                                                   */
/*                                                                           */
/*    Copydisk [FROM] drive [TO] drive [MULTI] [NOVERIFY]                    */
/*                                                                           */
/*    where the drives specified must both be 'trackdisk.device'             */
/*    controlled floppy drives (but may be the same device). The MULTI       */
/*    switch requests that multiple copies of the source disk be made.       */
/*    The NOVERIFY switch will cause the copying to be performed faster,     */
/*    but will not check for errors on the destination disk.                 */
/*                                                                           */
/*  NOTES:                                                                   */
/*    This program requires ARP.library, which must be in your libs:         */
/*    directory.                                                             */
/*                                                                           */
/*    This program can be compiled with either the Lattice                   */
/*    (version 5.02) or Manx (version 3.6a) compilers.                       */
/*    To compile/link with Lattice, use the following:                       */
/*          lc -O -cf copydisk                                               */
/*          blink from lib:arpc.o copydisk.o                                 */
/*                to   copydisk                                              */
/*                lib  lib:arp.lib lib:lc.lib lib:amiga.lib                  */
/*                SMALLCODE NODEBUG                                          */
/*    To compile/link with Manx, use:                                        */
/*          cc -o copydisk.o copydisk                                        */
/*          ln -o copydisk -larp -lc                                         */
/*                                                                           */
/*****************************************************************************/
#include <exec/types.h>
#include <exec/memory.h>
#include <exec/ports.h>
#include <exec/io.h>
#include <exec/exec.h>
#include <devices/trackdisk.h>
#include <libraries/dos.h>
#include <libraries/dosextens.h>
#include <libraries/filehandler.h>
#include <libraries/arpbase.h>
#include <libraries/arpfunctions.h>
#include <stdio.h>

#ifdef LATTICE
#include <stdlib.h>
#include <string.h>
#include <dos.h>
#include <proto/exec.h>
#include <proto/intuition.h>
#include <proto/graphics.h>
#include <proto/dos.h>
#endif /* Lattice */

#ifdef AZTEC_C
#define __ARGS(a) ()
#include <functions.h>
struct IORequest *CreateExtIO();
void memcpy();
#define memcpy(to,from,lgth) movmem(from,to,lgth)
#endif /* Aztec */

/* The following variables support the ARP startup code, including the needs
 * of the builtin GADS() call contained in that startup.
 */

#define ARG_FROM    1
#define ARG_TO      2
#define ARG_MULTI   3
#define ARG_NVFY    4

char *CLI_Template = "FROM/A,TO/A,MULTI/S,NOVERIFY/S";

char *CLI_Help =
"\2331;32mFROM\2330;31m and \2331;32mTO\2330;31m must be \
floppy disk drives\n\
\2331;32mMULTI\2330;31m, if specified, requests multiple \
copies of the same disk\n\
\2331;32mNOVERIFY\2330;31m requests that the output disk \
is not validated after it is written\n";

/* Other global variables
 */

char c_rite[] =
    "\n\2332;33mCopydisk\2330;31m \251Copyright 1989 - Bob Rakosky\n\n";
char src_prompt[] =
    "Place \2331;32mSource\2330;31m disk in drive %s:\n";
char dst_prompt[] =
    "Place \2331;32mDestination\2330;31m disk in drive %s:\n";
char ret_prompt[] =
    "   \2332;33mand press RETURN\2330;31m (ctrl-C/RETURN to abort)";

/* The following define is the length of a track, in bytes */
#define FULL_TRACK (TD_SECTOR * NUMSECS)

WORD    icur_trk = 0;
WORD    ocur_trk = 0;
WORD    max_trk = 0;

struct IOStdReq *td_ireq;
struct IOStdReq *td_oreq;
struct MsgPort *td_port;

/* The following structure is used to maintain a linked-list
 * of tracks of data in memory
 */

struct cache {
    struct cache *next;
    UBYTE buf[FULL_TRACK];
}
     *trk_buf = NULL;

int   num_bufs = 0; /* the number of track buffers that we've allocated */

char *src_name,
     *dst_name;

ULONG idsk_unit = -1,
      odsk_unit = -1;

UBYTE tdin_open  = FALSE,
      tdout_open = FALSE,
      idev_owned = FALSE,
      odev_owned = FALSE,
      dos_disk = FALSE,
      do_verify = TRUE,
      do_multi = FALSE,
      disk_in_mem = FALSE,
      have_src_disk = FALSE;

char *td_buf = NULL;

char  inp_line[MaxInputBuf];

/* The following are the prototypes for all of the functions local to this
 * program. Note that the __ARGS() macro will expand two different ways for
 * Lattice and Manx. For the Lattice compiler, the argument types for the
 * functions are specified. Since the Aztec compiler (v3.6a) doesn't
 * support function prototypes, the __ARGS() macro expands to just the
 * function-specification parentheses ().
 */

int  do_copy        __ARGS( (void) );
int  init_devs      __ARGS( (void) );
int  valid_devs     __ARGS( (void) );
int  open_td        __ARGS( (void) );
int  inhibit_drives __ARGS( (int) );
int  read_track     __ARGS( (struct cache *) );
int  write_track    __ARGS( (struct cache *) );
void touch_root_blk __ARGS( (char *) );
void stop_drives    __ARGS( (void) );
void cleanup        __ARGS( (void) );

/*****************************************************************************/
/* Function: main()                                                          */
/*                                                                           */
/* Purpose:  Main function of program                                        */
/*                                                                           */
/* Inputs:   Command line arguments                                          */
/*                                                                           */
/* Outputs:  Program exit code                                               */
/*                                                                           */
/* Notes:                                                                    */
```

```
/*   Since this program uses the ARP startup code, the Arp.library is     */
/*   opened and the command line arguments are processed according to our */
/*   template by the startup code before control passes to the main()     */
/*   function.  Note that the builtin GADS() function to process the      */
/*   command line will cause the program to terminate before we get here, */
/*   if an invalid command line is entered (except in the case that the   */
/*   program is invoked with NO arguments), in which case argc will       */
/*   a value of 1.                                                        */
/*************************************************************************/

void main(argc,argv)
    int argc;
    char **argv;
{
    int ret;

    Printf(c_rite);

    if (argc == 1) {
        Printf("Usage: %s %s\n\n",argv[0],CLI_Template);
        Printf(CLI_Help);
        exit(10);
    }
        /* Save the pointers to the source/dest drive names */
    src_name = argv[ARG_FROM];
    dst_name = argv[ARG_TO];

    if (argv[ARG_MULTI])
    {
        do_multi = TRUE;
    }
    if (argv[ARG_NVFY])
    {
        do_verify = FALSE;
    }

    if (!init_devs())       /* if error during initialization */
    {
        cleanup();
        exit(20);
    }

    ret = do_copy();
    cleanup();
    exit(0);
}

/*************************************************************************/
/* Function: do_copy()                                                   */
/*                                                                       */
/* Purpose:  controls the high-level reading and writing of disks        */
/*                                                                       */
/* Inputs:   none                                                        */
/*                                                                       */
/* Outputs:  The (eventual) program return code                          */
/*                                                                       */
/* Notes:                                                                */
/*   Only checks for cntl-c interrupts when the user is being prompted.  */
/*************************************************************************/

int do_copy()
{
    struct cache *buf_ptr;

    while (TRUE)
    {
        if (!have_src_disk)
        {
          /* prompt for source disk to be inserted */
            Printf(src_prompt,src_name);
            if (odsk_unit != idsk_unit)     /* if not single-drive copy */
            {
                Printf(dst_prompt,dst_name);
            }
            Printf(ret_prompt);
            ReadLine(inp_line);
            if (CheckAbort(NULL))   /* check for cntl-c */
            {
                return(0);
            }
            have_src_disk = TRUE;
        }
        icur_trk = ocur_trk = 0;
        while (ocur_trk < max_trk)
        {
            if (!disk_in_mem)   /* if we haven't buffered the src disk yet */
            {
                buf_ptr = trk_buf;      /* point to first cache */
                while ((buf_ptr) && (icur_trk < max_trk))
                {
                    if (!read_track(buf_ptr))
                    {
                        Printf("Unable to read source disk\n");
                        return(10);
                    }
                    buf_ptr = buf_ptr->next;    /* next in linked-list */
                }
                if (num_bufs >= max_trk)
                {
                    disk_in_mem = TRUE;     /* we've cached the entire disk */
                }
            }
            stop_drives();                  /* turn the motors off */
            if (idsk_unit == odsk_unit)
            {
              /* single drive copy - can't have source
               * and dest both present
               */
                Printf(dst_prompt,dst_name);
                Printf(ret_prompt);
                ReadLine(inp_line);
                if (CheckAbort(NULL))
                {
                    return(0);
                }
                have_src_disk = FALSE;
            }
            buf_ptr = trk_buf;      /* point to first cache */
            while ((buf_ptr) && (ocur_trk < max_trk))
            {
                if (!write_track(buf_ptr))
                {
                    Printf("Unable to write destination disk\n");
                    return(10);
                }
                buf_ptr = buf_ptr->next;    /* next in linked-list */
            }
            stop_drives();                  /* motor off */
            if (ocur_trk < max_trk)     /* if we haven't written the */
            {                           /* entire disk               */
                if (idsk_unit == odsk_unit)
                {
                  /* single-drive copy */
                    Printf(src_prompt,src_name);
                    Printf(ret_prompt);
                    ReadLine(inp_line);
                    if (CheckAbort(NULL))
                    {
                        return(0);          /* cntl-c detected */
                    }
                }
                continue;
            }
        }
        Printf("\nDisk copied\n");
        if (!do_multi)
        {
            break;          /* exit the while(TRUE) if single copy */
        }
        if (disk_in_mem)
        {
            Printf(dst_prompt,dst_name);
            Printf(ret_prompt);
            ReadLine(inp_line);
            if (CheckAbort(NULL))
            {
                break;
            }
        }
    }
    return(0);
}

/*************************************************************************/
/* Function: init_devs()                                                 */
/*                                                                       */
```

```
/* Purpose:  Validates and initializes disk device(s) specified          */
/*           Allocates required buffers                                  */
/*                                                                       */
/* Inputs:   none                                                        */
/*                                                                       */
/* Outputs:  Boolean success indicator                                   */
/*                                                                       */
/* Globals:  Device unit numbers                                         */
/*           IORequest structures                                        */
/*           Cache buffers to hold as much of disk data as memory allows */
/*           Trackdisk IO buffer (in CHIP memory)                        */
/*                                                                       */
/* Notes:                                                                */
/*    High level driver for initialization/validation                    */
/*    Uses the ARP memory allocation routines to track our memory        */
/*      allocations so that they are automatically freed on program exit. */
/*************************************************************************/

int init_devs()
{
    long    totmem;
    int     i;

    struct cache *cache_ptr;

        /* Note that if the drives have been specified using a trailing
         * colon, as is the standard means of specifying a device name, this
         * colon is stripped off.  This is because the device name will be
         * compared with the device name as contained in the AmigaDOS system
         * list of devices, and the names in that list do _NOT_ contain the
         * colons.
         */
    i = strlen(src_name);
    if (src_name[i-1] == ':')
    {
        src_name[i-1] = '\0';
    }
    i = strlen(dst_name);
    if (dst_name[i-1] == ':')
    {
        dst_name[i-1] = '\0';
    }

    if (!valid_devs())      /* insure that the devices specified    */
    {                       /* are valid device names, and are      */
        return(FALSE);      /* actual 'trackdisk.device' devices    */
    }

    if (!open_td())         /* open the devices */
    {
        return(FALSE);
    }
        /* Keep the file system from accessing the drives */
    if (!inhibit_drives((int)TRUE))
    {
        return(FALSE);
    }

      /* NOTE that trackdisk requires it's buffer to be in CHIP memory
       */
    td_buf = (char *)ArpAllocMem(FULL_TRACK,
                            MEMF_CHIP | MEMF_PUBLIC |MEMF_CLEAR);
    if (!td_buf) {
        Printf("Not Enough Chip MEMORY to Proceed\n");
        return(FALSE);
    }

      /*  We want to allocate enough cache buffers to hold the entire source
       *  diskette in memory, if possible.  We will, however, insure that at
       *  least 64K of memory is left unallocated. This will insure that the
       *  system has enough memory to handle most transient requirements.
       */
    totmem = (long)AvailMem(0L);
    totmem -= 65536L;        /* leave at least 64K available to the system */

    if (totmem >= (sizeof(struct cache)<<1) ) /* enough for at least two */
    {
        totmem /= sizeof(struct cache);
        if (totmem > max_trk) totmem = max_trk;
        for (num_bufs = 0; num_bufs < totmem; num_bufs++)
        {
            cache_ptr = (struct cache *)
                    ArpAllocMem((long)sizeof(struct cache),(long)MEMF_CLEAR);
            if (!cache_ptr)
            {
                break;
            }
            cache_ptr->next = trk_buf;
            trk_buf = cache_ptr;
        }
    }
    return(TRUE);
}

/*************************************************************************/
/* Function: valid_devs()                                                */
/*                                                                       */
/* Purpose:  Verifies that the devices requested for source and          */
/*           destination are valid devices in the system, and are, in    */
/*           fact, controlled by trackdisk.device                        */
/*                                                                       */
/* Inputs:   Global: source-name, destination-name                       */
/*                                                                       */
/* Outputs:  Boolean success indicator                                   */
/*           Global: source/destination unit numbers                     */
/*                                                                       */
/* Notes:                                                                */
/*   This routine walks through the system-level linked list of device   */
/*   nodes.  It must do this with task-switching disables, to insure that */
/*   this chain doesn't change while we are inspecting it.               */
/*                                                                       */
/*   NOTE that the DosLibrary structure, which is used to locate the     */
/*   device node list, is normally pointed to by the standard DosBase    */
/*   pointer.  However, because we are using the ARP startup, DosBase    */
/*   points to ArpBase, and not all of the DosLibrary structure is valid. */
/*   We have to use the REAL DosBase pointer, which is conveniently saved */
/*   for us in the Arp Library structure.                                */
/*************************************************************************/

int valid_devs()
{
    char    nm_buf[23],
            dev_buf[64];
    int     err = FALSE;
    struct DosLibrary *real_dos;
    struct DeviceNode *dlist;
    struct RootNode *rn;
    struct DosInfo *di;
    struct FileSysStartupMsg *fs;

    Forbid();   /* insure that the linked-list won't change on us */

      /* NOTE: DosBase does not _really_ point to DosBase */
    real_dos = (struct DosLibrary *)ArpBase->DosBase;

    rn = (struct RootNode *)real_dos->dl_Root;
    di = (struct DosInfo *) BADDR(rn->rn_Info); /* DosInfo is a BCPL ptr */
    dlist = (struct DeviceNode *) BADDR(di->di_DevInfo);  /* first entry */

    while (dlist) {                     /* until we get to end-of-chain */
        if (dlist->dn_Type == DLT_DEVICE)
            /*  Note:  the DeviceNode list contains entries for
             *    devices, directories (logical assigns), and
             *    volumes.  We are only interested in the device entries.
             */
        {
            BtoCStr(nm_buf,dlist->dn_Name,23L); /* convert BCPL string */

            if ((Strcmp(src_name,nm_buf)==0) || /* does it match? */
                (Strcmp(dst_name,nm_buf) == 0))
            {
                fs = (struct FileSysStartupMsg *)
                            BADDR(dlist->dn_Startup);   /* it's a BPTR */

                BtoCStr(dev_buf,fs->fssm_Device,63L);   /* convert BCPL str */
                if (Strcmp(dev_buf,TD_NAME) == 0)   /* is it trackdisk? */
                {
                    if (Strcmp(src_name,nm_buf)==0)
                    {
                        idsk_unit = fs->fssm_Unit;  /* grab the real unit # */
                    }
                    if (Strcmp(dst_name,nm_buf)==0)
                    {
                        odsk_unit = fs->fssm_Unit;
                    }
                }
```

```
                else        /* the device is not trackdisk.device */
                {
                    Permit();       /* remember to enable multitasking */
                    Printf("Error: %s device not a floppy drive\n",
                        (Strcmp(src_name,nm_buf)==0) ? "Source" : "Destination");
                    return(FALSE);
                }
            }
        }
        dlist = (struct DeviceNode *) BADDR(dlist->dn_Next); /* next in chain */
    }
    Permit();               /* multitasking OK again */

      /* NOTE: the device unit numbers are initialized to -1. If that value
       * hasn't changed, then no matching entry was found for that device
       * name in the list.
       */
    if (idsk_unit == -1L)
    {
        Printf("Error: Source device not found\n");
        err = TRUE;
    }
    if (odsk_unit == -1L)
    {
        Printf("Error: Destination device not found\n");
        err = TRUE;
    }
    if (err)
    {
        return(FALSE);
    }
    return(TRUE);
}

/*************************************************************************/
/* Function: open_td()                                                   */
/*                                                                       */
/* Purpose:  Open the trackdisk.device units, allocate the IO request    */
/*           structures, get/validate disk information (total number of  */
/*           tracks).                                                    */
/*                                                                       */
/* Inputs:   Global: unit numbers (src and dest)                         */
/*                                                                       */
/* Outputs:  Boolean success indicator                                   */
/*           Global:  input and output IO request structures             */
/*                    max-track                                          */
/* Notes:                                                                */
/*    We only will perform the copy if the source and destination devices */
/*    have the same maximum number of tracks.  We won't, for example,    */
/*    copy from a 5-1/4" floppy to a 3-1/2" drive.                       */
/*************************************************************************/

int open_td()
{
    td_port = (struct MsgPort *) CreatePort(0,0);
    if (!td_port)
    {
        Printf("Unable to allocate a Port/Signal Bit\n");
        return(FALSE);
    }
    td_ireq = CreateStdIO(td_port);
    if (!td_ireq) {
        Printf(" Not Enough Memory to Proceed\n");
        return(FALSE);
    }
    if (OpenDevice(TD_NAME,idsk_unit,(struct IORequest *)td_ireq,0L)) {
        Printf(" Can\'t allocate Source Device\n");
        return(FALSE);
    }
    tdin_open = TRUE;

    td_oreq = CreateStdIO(td_port);
    if (!td_oreq) {
        Printf(" Not Enough Memory to Proceed\n");
        return(FALSE);
    }
    if (OpenDevice(TD_NAME,odsk_unit,(struct IORequest *)td_oreq,0L)) {
        Printf(" Can\'t allocate Target Device\n");
        return(FALSE);
    }
    tdout_open = TRUE;

    td_ireq->io_Command = TD_GETNUMTRACKS;
    DoIO((struct IORequest *)td_ireq);
    max_trk = td_ireq->io_Actual;

    if (odsk_unit != idsk_unit) {
        td_oreq->io_Command = TD_GETNUMTRACKS;
        DoIO((struct IORequest *)td_oreq);
        if (max_trk != td_oreq->io_Actual)
        {
            Printf("Incompatible devices - different track numbers\n");
            return(FALSE);
        }
    }
    return(TRUE);
}

/*************************************************************************/
/* Function: inhibit_drives()                                            */
/*                                                                       */
/* Purpose:  Either dis-allow the file-system from using the             */
/*           drives, or re-allow standard file-system access             */
/*                                                                       */
/* Inputs:   Boolean - TRUE == inhibit file-system access                */
/*                     FALSE == allow file-system access                 */
/*           Global: device names                                        */
/*                                                                       */
/* Outputs:  Boolean success indicator                                   */
/*                                                                       */
/* Notes:                                                                */
/*   Uses DOS packet-level IO.                                           */
/*   When the devices are 'inhibited', they are treated as               */
/*     "Not a DOS disk", and will show as DFn:BUSY on the WB screen.     */
/*************************************************************************/

int inhibit_drives(i_flag)
    int i_flag;
{
    char dev_nam[10];
    int     i;
    long    pkt_args[7];
    struct MsgPort *idev_proc,
                   *odev_proc;

    sprintf(dev_nam,"%s:",src_name);    /* insure the name ends with : */
        /* Get the address of the process's MsgPort structure for
         * the device driver.
         */
    idev_proc = (struct MsgPort *)DeviceProc(dev_nam);
    if (!idev_proc) {
        Printf(" Can\'t locate Source Device's process\n");
        return(FALSE);
    }
      /* If the input and output devices are the same, the processes
       * are the same.  Otherwise, repeat for the output device
       */
    if (odsk_unit == idsk_unit) {
        odev_proc = idev_proc;
    }
    else {
        sprintf(dev_nam,"%s:",dst_name);
        odev_proc = (struct MsgPort *)DeviceProc(dev_nam);
        if (!odev_proc) {
            Printf(" Can\'t locate Target Device's process\n");
            return(FALSE);
        }
    }
    for (i=0;i<7;i++)
    {
        pkt_args[i] = 0;    /* initialize the packet argument array */
    }
    pkt_args[0] = i_flag;   /* First argument is a Boolean value */

    if (!SendPacket(ACTION_INHIBIT,pkt_args,idev_proc))
    {
        Printf("Unable to inhibit source drive\n");
        return(FALSE);
    }
    if (i_flag)
    {
        idev_owned = TRUE;
    }

    if (idsk_unit != odsk_unit)     /* only if two devices */
    {
        pkt_args[0] = i_flag;
        if (!SendPacket(ACTION_INHIBIT,pkt_args,odev_proc))
        {
            Printf("Unable to inhibit destination drive\n");
```

```
            return(FALSE);
        }
        if (i_flag)
        {
            odev_owned = TRUE;
        }
    }
    return(TRUE);
}

/*************************************************************************/
/* Function: read_track()                                                */
/*                                                                       */
/* Purpose:  reads one track from the input disk                         */
/*                                                                       */
/* Inputs:   cache buffer pointer into which to copy the data            */
/*                                                                       */
/* Outputs:  Boolean success indicator                                   */
/*           Global: Boolean DOS disk indicator (track 0 only)           */
/*                                                                       */
/* Notes:                                                                */
/*   We will retry the read up to 3 times                                */
/*                                                                       */
/*   For track 0 only, we will inspect the first 4 bytes of the first    */
/*   sector to determine whether the disk we are reading is a DOS disk.  */
/*   We will have to 'twiddle' with DOS disks when writing to insure we  */
/*   don't create an 'identical' disk.                                   */
/*************************************************************************/

int read_track(cbuf)
    struct cache *cbuf;
{
    long i;
    int retry;

    Printf(" Reading   Cylinder %ld Head %ld     \r",
            (long)(icur_trk >> 1),
            (long)(icur_trk & 1));
    for (retry=0;retry<3;retry++)        /* up to 3 retries */
    {
        td_ireq->io_Command = CMD_READ;
        td_ireq->io_Length = FULL_TRACK;
        td_ireq->io_Offset = (long)(icur_trk * FULL_TRACK);
            /* NOTE that trackdisk reads must be to a buffer that is in CHIP
             * memory.  Consequently, we can't read directly into our cache
             * buffer.
             */
        td_ireq->io_Data = (APTR) td_buf;
        i = DoIO((struct IORequest *)td_ireq);
        if (!i)     /* success */
        {
            if (icur_trk == 0)
            {
                if ((Strncmp(td_buf,"DOS\000",4L) == 0) ||
                    (Strncmp(td_buf,"DOS\001",4L) == 0) )
                {
                    dos_disk = TRUE;
                }
                else
                {
                    dos_disk = FALSE;
                }
            }
            icur_trk++;
            memcpy(cbuf->buf,td_buf,(int)FULL_TRACK);   /* copy to cache */
            return(TRUE);
        }
    }
    return(FALSE);
}

/*************************************************************************/
/* Function: write_track()                                               */
/*                                                                       */
/* Purpose:  Writes one track of data to the destination device         */
/*           and optionally reads it back in to verify the write         */
/*                                                                       */
/* Inputs:   cache buffer                                                */
/*                                                                       */
/* Outputs:  boolean success indicator                                   */
/*                                                                       */
/* Notes:                                                                */
/*   Uses the 'format' command, rather than the 'write' command, as this */
/*   is faster, and works whether the disk is formatted or not. It is,   */
/*   however, a little more prone to errors, which is why the 'verify'   */
/*   option is recommended.                                              */
/*************************************************************************/

int write_track(cbuf)
    struct cache *cbuf;
{
    long i;
    int  retry;
    static char *wr_prot_pr1 =
        "\nDisk in drive %s: is \23332;32mWRITE PROTECTED\23330;31m\n";
    static char *wr_prot_pr2 =
    "   \2332;33mPress RETURN to continue\2330;31m (ctrl-C/RETURN to abort)";

    Printf(" Writing   Cylinder %ld Head %ld     \r",
            (long)(ocur_trk>>1),
            (long)(ocur_trk & 1));

      /* NOTE:  AmigaDOS really doesn't like two DOS disks in the system
       * that are absolutely identical.  This will cause the system to lock
       * up or GURU if both disks are present at the same time, if the file-
       * system has access to the disks.  We will 'tweak' the root block of
       * the output disk, to avoid the 'identical disk' syndrome.
       */
    if (dos_disk)
    {
        if (ocur_trk == (max_trk >> 1)) /* root block is the middle */
        {                               /* sector of the device     */
            touch_root_blk(cbuf->buf);
        }
    }
      /* We have to copy the data to our buffer in CHIP memory
       * for the trackdisk.device to be able to access it.
       */
    memcpy(td_buf,cbuf->buf,(int)FULL_TRACK);

    for (retry=0;retry<3;retry++)   /* retry up to 3 times */
    {
        td_oreq->io_Command = TD_FORMAT;
        td_oreq->io_Length = FULL_TRACK;
        td_oreq->io_Offset = (long)(ocur_trk * FULL_TRACK);
        td_oreq->io_Data = (APTR) td_buf;

        i=DoIO((struct IORequest *)td_oreq);
        if (i)      /* error occurred - retry */
        {
            if (td_oreq->io_Error == TDERR_WriteProt)
            {
                stop_drives();
                Printf(wr_prot_pr1,dst_name);
                Printf(wr_prot_pr2);
                ReadLine(inp_line);
                if (CheckAbort(NULL))   /* check for cntl-c */
                {
                    break;
                }
            }
            continue;
        }
        if (do_verify)
        {
            Printf(" Verifying Cylinder %ld Head %ld     \r",
                    (long)(ocur_trk>>1),
                    (long)(ocur_trk & 1));
            td_oreq->io_Command = CMD_READ;
            td_oreq->io_Length = FULL_TRACK;
            td_oreq->io_Offset = (long)(ocur_trk * FULL_TRACK);
            td_oreq->io_Data = (APTR) td_buf;

            i=DoIO((struct IORequest *)td_oreq);
            if (i)      /* can't read track - retry the write */
            {
                continue;
            }
            i = memcmp(td_buf,cbuf->buf,(int)FULL_TRACK);
            if (i)      /* compare failed */
            {
                continue;
            }
        }
        ocur_trk++;
        return(TRUE);
    }
    return(FALSE);
}
```

```
/*************************************************************************/
/* Function: touch_root_blk()                                            */
/*                                                                       */
/* Purpose:  'Tweaks' the root block, adjusting the 'date created' field, */
/*           so that this disk won't be identical to the disk we read.   */
/*                                                                       */
/* Inputs:   Pointer to the root block image                             */
/*                                                                       */
/* Outputs:  none (Root block image is updated)                          */
/*                                                                       */
/* Notes:                                                                */
/*   We verify that the block passed is indeed the root block.           */
/*                                                                       */
/*   The AmigaDOS Tech. Ref. Manual documents the format of the root     */
/*   block in terms of long-words.  The values used in this routine match */
/*   that documentation.                                                 */
/*                                                                       */
/*   The checksum for the block has to be re-calculated.  The checksum is */
/*   the value that causes the sum of all the long-words in the sector   */
/*   (including the checksum) to total a value of 0.                     */
/*************************************************************************/

void touch_root_blk(blk)
    char *blk;
{
    long *lptr;
    long max_lwords;
    long chksum;
    int i;

    lptr = (long *)blk;     /* treat block as array of longs */
    max_lwords = (TD_SECTOR >> 2);

    if ((lptr[0] != 2) ||
        (lptr[1] != 0) )        /* it's not the root block ??? */
    {
        return;
    }
    DateStamp(&lptr[max_lwords-7]);
    chksum = lptr[5] = 0;       /* initialize checksum to 0 */
    for (i=0;i<max_lwords;i++)
    {
        chksum += lptr[i];
    }
    lptr[5] = (long) (0 - chksum);
}

/*************************************************************************/
/* Function: stop_drives()                                               */
/*                                                                       */
/* Purpose:  Insures that the drive(s) motors are turned off             */
/*                                                                       */
/* Inputs:   none                                                        */
/*                                                                       */
/* Outputs:  none                                                        */
/*                                                                       */
/* Notes:                                                                */
/*   The trackdisk.device will always turn a drive's motor on when it is */
/*   needed (for a read or write), but does NOT turn the motor off.      */
/*   There is no harm done in attempting to turn off the motor if it is  */
/*   already off, so we don't really need to keep track of the motor's   */
/*   state, just turn it off when we want to insure it's off.            */
/*************************************************************************/

void stop_drives()
{
    td_ireq->io_Command = TD_MOTOR;     /* insure motor off */
    td_ireq->io_Length = 0;
    DoIO((struct IORequest *)td_ireq);
    if (odsk_unit != idsk_unit)
    {
        td_oreq->io_Command = TD_MOTOR;     /* insure motor off */
        td_oreq->io_Length = 0;
        DoIO((struct IORequest *)td_oreq);
    }
}

/*************************************************************************/
/* Function: cleanup()                                                   */
/*                                                                       */
/* Purpose:  Undoes what we did                                          */
/*                                                                       */
/* Inputs:   none                                                        */
/*                                                                       */
/* Outputs:  none                                                        */
/*                                                                       */
/* Notes:                                                                */
/*   returns resources to the system.  NOTE that since our memory        */
/*   allocations have all been used with the Arp allocation routines,    */
/*   they will automatically be freed during normal exit() processing    */
/*   (when the Arp.library is closed).                                   */
/*************************************************************************/

void cleanup()
{
    if (tdin_open && tdout_open)
    {
        stop_drives();
    }
    if (tdin_open) {
        CloseDevice((struct IORequest *)td_ireq);
        tdin_open = FALSE;
    }
    if (tdout_open) {
        CloseDevice((struct IORequest *)td_oreq);
        tdout_open = FALSE;
    }
    if (idev_owned || odev_owned)
    {
        inhibit_drives((int)FALSE);
        idev_owned = odev_owned = FALSE;
    }
    if (td_ireq) {
        DeleteStdIO(td_ireq);
        td_ireq = NULL;
    }
    if (td_oreq) {
        DeleteStdIO(td_oreq);
        td_oreq = NULL;
    }
    if (td_port) {
        DeletePort(td_port);
        td_port = NULL;
    }
    /*  Normally we would free up our buffer allocations here
     *  but since we allocated them using the ArpAllocMem function
     *  they will auto-magically be freed when we exit()
     */
}

#ifdef AZTEC_C

/*************************************************************************/
/* Function:  memcmp()                                                   */
/*                                                                       */
/* Purpose:   compares two blocks of memory                              */
/*                                                                       */
/* Inputs:    pointers to the memory, length of the block to compare     */
/*                                                                       */
/* Outputs:   negative if first block is less than second                */
/*            0 if the two blocks are identical                          */
/*            positive if first block is greater than second             */
/*                                                                       */
/* Notes:                                                                */
/*   This is a standard library function for Lattice, but the Manx       */
/*   compiler library didn't seem to have an equivalent function.        */
/*************************************************************************/

int memcmp(s1,s2,lgth)
    register char *s1, *s2;
    register int lgth;
{
    register int res;

    while (--lgth)
    {
        if (res = (*(s2++) - *(s1++)))
        {
            return(res);
        }
    }
    return(0);
}

#endif  /* if AZTEC */
```

A Walk In The Binary Forest

The theory and practice of three tree types

by Eric Giguère
Copyright © 1989 Eric Giguère

Eric Giguère is a computer science student at the University of Waterloo, and has been involved in the design of an ANSI-conformant C compiler with the Computer Systems Group.

A recent project of mine involved optimizing a C compiler for speed. The code generator was as fast as it would get, but the front end (responsible for reading in the C source program and converting it into a form usable by the code generator) needed some work. In particular, symbol table management was much too slow.

The symbol table in a compiler is responsible for keeping track of the symbols - variables, macros, type names - used by a program. The code generator must make constant references to the symbol table to locate needed information which is why look-up speed is important - a symbol should be located in the table with as few searches as possible. A symbol table for a C compiler must also handle the complicated scoping and naming rules of the language, and so a balance must be found. My solution was to use *binary trees*, which I then expanded into a complete library of C routines to implement several types of trees. I refer to the library as the *Tree Package*, or simply *TreePak*.

This article has two main components. The first is an introduction to trees in general and three types of binary trees: simple, splay and AVL. The second component is a short tutorial on using the *TreePak* routines. The *TreePak* code is in the public domain and is included on the diskette for this issue. As a whole, *TreePak* is about two thousand lines long, split across five modules. Also included on the diskette is a full set of documentation files. A complete reference manual in DVI format has also been included for *AmigaTeX* users to print out.

Trees

The *tree* is a common structure in nature that allows plants to spread their leaves over a large area and gather as much energy as possible. The structure has come to be used in other contexts as well, such as in a genealogical tree describing a person's ancestry. The tree is also a useful data structure in computer science.

A tree is made up of *nodes* and *links*. The nodes hold the data for the tree and the links connect the nodes together as a tree. The links are best thought of as 'unidirectional' - having one direction. The basic idea behind the tree structure is that every node has at most one link pointing to it, though each can be the origin of many links. If a node has a link pointing to another node, it is referred to as the *parent node* of the second node, the *child node*. A node has at most one parent but possibly many children.

The directories on your disks are organized as trees. A **dir** command from the CLI for my current disk shows:

```
Manx3.6a (dir)
Work (dir)
```

Typing **cd Work** and **dir** again shows:

```
Trees (dir)
Bison (dir)
```

and a list of files. The two directories 'Trees' and 'Bison' are subdirectories, the children of 'Work'. 'Work' is the parent directory. A graphical representation might be:

```
     Work
    /    \
Trees    Bison
```

which makes the tree structure explicit. Childless nodes are called *leaf nodes*. Each node also has a *level* or *height* in the tree, and the highest node is referred to as the *root* node.

How are trees implemented on a computer? The simplest method is to define a structure type for each node:

```
struct tree_node
   {
      struct tree_node *child1, *child2, ... ;
      /* node data goes here */
   };
```

The structure for a node holds both the node data and pointers to the node's children, if any. This implementation is similar in many ways to the use of linked lists.

Binary trees

The more children a node has, the more complicated the tree. *TreePak* is restricted to a class of trees called binary trees. Each node in a *binary tree* has at most two children, referred to as the *left* and *right* children.

Binary trees can be used to hold data in a form suitable for quick searches. Each node contains a value called a *key*. The key is usually an integer or a string but can be any combination of data. The node may contain other data as well, but it is the key that determines the ordering of the nodes in the binary tree, and it is the key that is used in searching for particular nodes.

The basic premise of a binary tree is this: for each node, the left child (if any) has a *smaller* key and the right child (if any) has a *larger* key. This also implies that all the children of the left child must be smaller than the key, and all their children, and so on, and that the reverse holds for the children of the right child. *Each key in the subtree rooted at the left child (the left descendants of the node) is smaller, and each key in the subtree rooted at the right child (the right descendants) is larger.* Note that the definition of 'smaller' and 'larger' is flexible. In most cases the natural order is used - 3 is less than 5, "abc" is less than "xyz" - but the ordering method doesn't matter as long as it's consistent. A simple binary tree is shown in Figure 1, with integers used as keys. One problem with binary trees is that nodes with equal keys may confuse matters somewhat. *TreePak* avoids this by disallowing duplicate keys in the same tree.

Operations

Three basic operations can be performed on binary trees: insertion, deletion and search. The search operation is the simplest:

1. Let the root node be the current node. If no root node exists then the tree is empty, exit the search and return FALSE.

2. Compare the current node's key with the search key. If they are equal, exit the search and return TRUE.

3. If the search key is smaller than the current node's key, let the new current node be the left child of the current node. If no left child exists, exit the search and return FALSE, otherwise go to step 2.

4. The search key must be larger than the current node's key, so let the new current node be the right child of the current node. If no right child exists, exit the search and return FALSE, otherwise go to step 2.

This binary search algorithm is very efficient because it weeds out large groups of nodes with every comparison. It is no worse, and usually quite a bit better, than performing a linear search on a linked list.

The insertion operation adds new nodes to a binary tree. It is similar to the binary search algorithm. First, we search the tree for the node we want to insert. If we find a node with the same key, the operation returns an error. Otherwise, the search fails, and we insert the new node as the left or right child (whichever is appropriate) of the last node we visited. Inserting the value 25 into the tree of Figure 1 would make it the right child of 33.

Deleting a node is more complicated because the ordering of the tree must be preserved. Thus we search for the next largest key in the tree and move it into the deleted node's position. Deleting 8 from the tree of Figure 1 would yield the tree of Figure 2, with 15 as the new root. Notice that the binary ordering of the tree has been preserved.

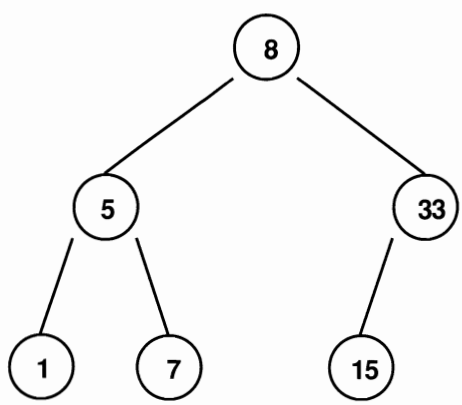


Figure 1: A simple binary tree

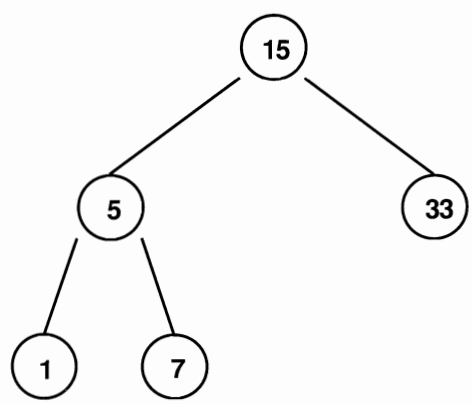


Figure 2: The tree of Figure 1 after a deletion

TreePak also provides a fourth operation that removes all the nodes of a tree; this is often called 'burning' the tree.

The trees of Figures 1 and 2 might be implemented using the C structure

```
struct tree_node
   {
      struct tree_node *left, *right;
      int              key;
   };
```

with NULL pointers indicating that the node has one or no children, as appropriate.

Splay trees

Simple binary trees have the problem that nodes will always be inserted near the bottom of the tree, leading to longer search times for the most recently inserted nodes. Moreover, simple binary trees can become horribly unbalanced. Inserting the sequence 1, 2, 3, 4, 5 into an initially empty tree yields a tree where each node has only a right child.

A *binary splay tree* (or simply *splay tree*) is one way to minimize these problems. A splay tree minimizes the search time for a particular node by reordering the binary tree to move the most frequently used nodes closer to the root of the tree. This is referred to as *splaying* the tree. Each time a node is accessed, it is moved to the root of the tree using a series of node rotations. The resulting tree is still a binary tree and is particularly suited to situations where only a small number of nodes are accessed with any frequency. A splay tree is one example of an *adaptive* searching method.

AVL trees

A more complex type of binary tree is the *height-balanced* or *AVL* tree. (The name AVL is an acronym for its inventors.) An AVL tree is a binary tree in which the *difference* of the heights of the left and right subtrees of any node is -1, 0 or +1. The trees in Figures 1 and 2 are both AVL trees. The node 8 in Figure 1 has a left subtree of height 2 and a right subtree of height 2, hence the difference is 0. The node 33 has a left subtree of height 1 and a right subtree of height 0, hence the difference is 1 (or -1, depending which way you look at it).

Figure 3 shows an example of a non-AVL tree - the height constraint is violated at node 25. AVL trees require more work to maintain because inserting or deleting a node may leave a non-AVL tree, requiring one or more transformations to return it to AVL form.

Its advantage over simple - and splay - binary trees is that an AVL tree has a lower upper limit on the number of searches required to find a node. In fact, it approximates an optimal search tree and is useful in situations where all the nodes in the tree are of equal search probability.

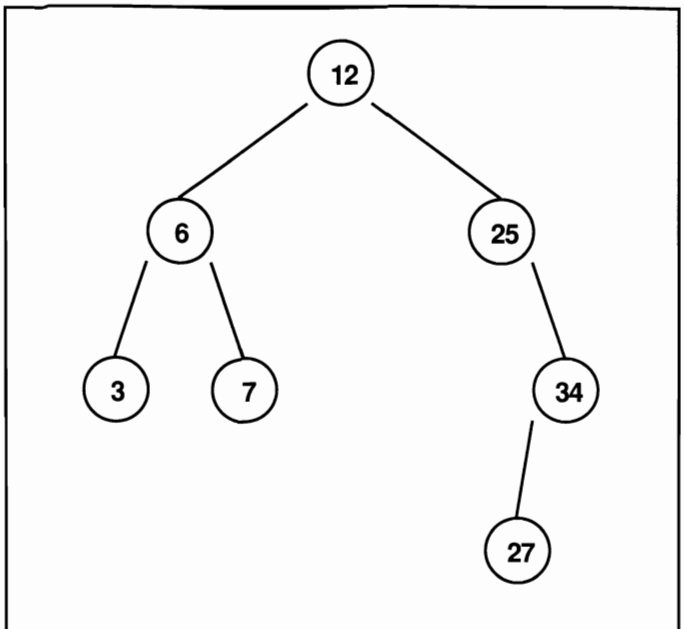


Figure 3: A non-AVL binary tree

TreePak

The code in *TreePak* is divided into five stand-alone modules. Three of these are sets of routines to implement binary, splay and AVL trees. The remaining two modules are used to display binary trees on an ASCII device in a semi-graphical form. The code is fully commented and, each module includes an example program that demonstrates its use.

One problem with implementing a general-purpose binary tree library lay in the fact that the types of the node keys were unknown. To be truly useful, the library should not be restricted to handling integer keys. Thus the routines have been written without any direct reference to the keys - the user of a library must supply a comparison function for use in searching a tree, as we'll see below.

Defining the nodes

Each of the three sets of C routines - tree.c, splay.c and avl.c - has an accompanying header file that must be included by any C file that uses the library. The header files declare the proper prototypes and an abstract data type that must be found at the start of every node. This data type contains all the information (except the key) needed by the *TreePak* routines.

A simple binary tree or splay tree like the ones we've seen in the figures could be defined by the following sequence:

```
#include "tree.h" /* or "splay.h" */

struct my_node
   {
      tree_node data;
      int       key;
   };
```

An AVL tree uses a different data type:

```
#include "avl.h"

struct my_node
   {
      avl_node data;
      int      key;
   };
```

Under no circumstances should the contents of the first member of your node structure be modified by any of your code.

Note that though both simple and splay binary trees use the same data structure, they use different routines that should not be mixed. And *never* use any of the routines in tree.c or splay.c on an AVL tree.

The comparison function

Inserting or searching a binary tree requires that node keys be compared. You must provide the *comparison function* for this because the library routines know nothing about the keys. The comparison function behaves much like your compiler's *strcmp()* routine: it takes two pointers to the nodes being compared and returns a negative, zero, or positive value depending if the first key is smaller than, equal to, or larger than the second key.

A simple comparison function for the data type we declared above would be:

```
int my_cmp( a, b )
   my_node *a, *b; /* the nodes being compared */
   {
      return( a->key - b->key );
   }
```

The comparison function should be altered to suit whatever keys you have defined. The address of this function is then passed to the insert and search routines whenever they are called (see below).

Inserting

The insertion routine for simple binary trees is called *tree_insert* and takes three parameters: the node to be inserted, the root of the tree, and the comparison function.

The prototype looks like:

```
int tree_insert(tree_node  *ins_key,
                tree_node **root,
                int       (*cmp)() );
```

Note that only pointers are passed. The *address* of a root pointer must be passed as the second parameter because the root might change with the insertion.

The following code fragment illustrates how to use the insertion routine using the data type and comparison function we defined previously:

```
my_node *new, *root, dummy;

root = NULL;
new  = malloc( sizeof( struct my_node ) );
new->key = 5;

tree_insert( new, &root, my_cmp );
```

tree_insert returns TRUE if the insertion was successful, FALSE otherwise. The equivalent routines for splay and AVL trees are *splay_insert* and *avl_insert*.

Searching

Searching is done with the *tree_find* routine, or its counterparts *splay_find* and *avl_find*.

Its prototype is similar to that for the insertion routine:

```
tree_node *tree_find(tree_node  *dummy,
                     tree_node **root,
                     int       (*cmp)() );
```

The first parameter is a dummy node whose key matches the key of the node you wish to find in the tree. The other two parameters are identical to those in the insertion routine:

```
dummy.key = 5;

if( ( new = tree_find( &dummy, &root, my_cmp ) )
      == NULL )
   puts( "node not found" );
```

tree_find returns a pointer to the node if the search is successful, NULL otherwise.

Deleting

The deletion routine *tree_delete* has the prototype:

```
tree_node *tree_delete( tree_node  *node,
                        tree_node **root );
```

The first parameter is the node to be deleted, typically found using the *tree_find* routine, and the second parameter is the root pointer.

```
free( tree_delete( new, &root ) );
```

tree_delete returns the original pointer to the deleted node. This is necessary because the deletion routine only removes the node from the tree. Deallocation (such as by the *free()* routine) and clean-up must be done by the user. The counterparts to *tree_delete* are *splay_delete* and *avl_delete*.

Burning

A whole tree can be removed by calling the *tree_burn* routine with two parameters:

```
void tree_burn( tree_node  *root,
                void       (*fp)() );
```

The first parameter is the root of the tree (note that only a simple pointer is required here, not a pointer to a pointer) and the second is a pointer to a deallocation routine. This deallocation routine will be called for each node in the tree after its removal.

```
tree_burn( root, free );
```

A NULL pointer can be passed instead of a function address, in which case no deallocation routine is called.

Printing a tree

TreePak also includes two routines for printing binary trees. These routines are not very sophisticated and should be used for debugging only. The first is *h_pr_tree*, found in hprint.c; it is used to print a binary tree horizontally across the screen.

The second is *v_pr_tree*, which prints a tree in the more usual vertical fashion. Both take a number of parameters to be used in controlling the printing: see the examples on diskette and the on-line documentation. Output is directed to *stdout*, which is usually the current console window.

Final notes

The examples included with the source code should give you more than enough information on how to use the *TreePak* library. Modifying the library code, particularly the AVL routines, is not recommended because several of them are quite complicated.

Binary trees are not necessarily the answer to all your data organization problems, but they can be very useful in certain situations. With AVL trees, I was able to improve the compiler's performance by 30 per cent. Not too shabby!

References

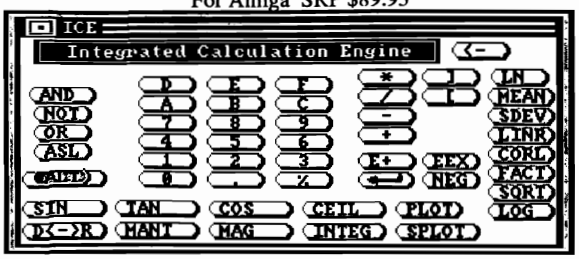
Donald Knuth, *The Art of Computer Programming*, Vols. 1 and 3.

Edward Reingold and Wilfred Hansen, *Data Structures in Pascal.*

Daniel Stubbs and Neil Webre, *Data Structures with Abstract Types and Modula-2.*

I.C.E.

Integrated Calculation Engine
For Amiga SRP $89.95



Who needs it?

Students Programmers Scientists Teachers Financiers Statisticians Businessmen Accountants Researchers Technicians Realtors Leasors Rental agents Contractors Engineers Mathematicians Numbercrunchers Everywhere!

Why?

I.C.E. makes number work easy! I.C.E. is a programmable calculator construction set. Create calculators designed to do the jobs you need done. Using the mouse, select the buttons (functions) you want for your customized calculators. Add the graphic plot &print functions, choose the layout and number system you find comfortable, bind macro programs to a single key for truly powerful versatility. Then use your I.C.E. creations with other programs like spreadsheets and databases. I.C.E. is not copy protected and multitasks beautifully! Now you can have programmable calculators with the features you choose. I.C.E. calculators include an integral program editor which allows you to create complex formulae from over fifty available functions.

KING PUBLISHING

921-32nd Street Sioux City IA 51104 Phone 712-252-4604

FREE: Amiga Demos

Have you ever wished you could view a program before you purchased it? Do artist conceptions and written descriptions really give you a good idea about the program you are buying? No!

We will send you actual manufacturers demonstration programs for you to view, on your own computer, in your own time. The gimmick is: if you like the program, buy it from us! We're even competitively priced!

Send us your name and address, we'll send you an ***Amiga Demo FREE!***

We ship anywhere in the World!

Forethought
Innovatiove Software Distribution
P.O. Box 1221
Lewiston, NY 14092

Programming In M2AMIGA

VectorFonts for graphic lettering

by Anthony Bryant
Copyright © 1988 A. Bryant

This is part of a series of articles on using the Modula 2 programming language, specifically the M2AMIGA Development System from Interface Technologies.

Writing text on an Amiga screen invariably means plotting text on a hi-res bitmap. Consequently, there are several methods of producing textual displays available to the programmer. Plotting big text on the screen implies either selecting a large font from disk, or enlarging a small style font, bit by bit. But the very nature of scaling bitmapped fonts to produce larger characters produces blocky-looking letters.

The ever-resourceful programmer might decide instead to 'paint' individual big letters by using a painting program and then storing them for later use in a program. Painting an entire font though, would seem impractical.

Another method of writing big text on the screen involves just that – using the Amiga's graphics functions to actually Draw() the letters one at a time as needed from a font of big letters. And drawing letters, composed only of line segments drawn on the screen, also allows a programmer to easily scale and rotate the text, allowing even bigger letters and permitting text to be skewed at an arbitrary angle. This article details the design and construction of a so-called 'VectorFont' and the associated drawing routines in M2AMIGA.

Designing letters

With the appropriate set of line segments (vectors), virtually any character shape can be drawn. The characters do not necessarily have to look like the standard ASCII character set, either. Consider the small set of characters that would be needed to draw musical notes, for example. What's involved in such a process? First we need to design a font format containing all of the instructions needed to draw each character. Each character must be allowed to be just as complex as it needs to be, so the drawing instructions should be flexible. Here we have total freedom to define our own format to specify what line segments or vectors should be drawn from what starting and ending coordinates. Another thing to realize is that the data will need to define these coordinates relative to the previous coordinates. By specifying the next point based on the previous point, the character can be drawn anywhere on the screen.

One way we could define the format of the data is to specify each vector with two pairs of x- and y-relative coordinates. A single signed byte could be used for each coordinate, which would represent a value from 127 to -128. Thus, four bytes would be required for each vector in the data. By specifying that each point continue from the previous point, we eliminate the need to specify a new beginning point every time. The problem is that there must be some way of moving to the next point without drawing, so that a vector can be specified using just one pair of relative coordinates, rather than two. For this to be an advantage, a high percentage of the vectors would need to draw from the end of the previous vector. When creating characters from vectors, I believe this will generally be true.

As well as signalling a move instead of a draw, we also need a way to signal the end of drawing a character, since characters will be composed of a varying number of vectors depending on complexity. One way to do this is to use one of the coordinate values to signal a special instruction. The other coordinate could be used to indicate which special instruction. These special instructions will then require additional bytes.

Now we are ready to get down to specifics: Let the data for each character comprise successive x- and y-relative coordinates or 'vector bytes', and let the xy pair vector bytes range from 127 to -127. I don't believe we will exceed this range, but if we do we could arrange another special instruction to use two successive bytes per value. So, let x = -128 signal a special instruction, in which case, y = 0 means the end of drawing and y = 1 means a move instead of draw. If we need any further special instructions in the future that we haven't thought of, we can use y = 2, y = 3, etc.

Experience with drawing characters shows that some characters take many vectors to draw and some, like 'i', take very few. Also, large characters definitely look better if the spacing between letters is proportional. One way to offset proportionally-drawn characters is first to define relative coordinate x = 0

as the centreline of the character, and then to specify a left edge (to the left of the centreline) as a negative value, and a right edge (to the right of the centreline) as a positive value. A character then starts all drawing relative to the left edge, and at the end of drawing moves to the right edge, proportionally-spaced and ready to start the next character.

Let's define the first two signed bytes in each character's data as the 'leftEdge' and 'rightEdge' values. These two x-relative coordinates will be read by our drawing routine before the xy pairs. Let's also look at the y-relative coordinates. Just like a regular bitmap font, our vector font also needs a baseline. Let y = 0 be the baseline; above the baseline are negative values, and below the baseline are positive values. So, relatively speaking, xy pair (0,0) is at the centre of our character grid. As it happens, this is also very convenient as a centre point for additional character manipulations! Proper x and y scaling and rotation transformations performed relative to the centre (0,0) produce the best results.

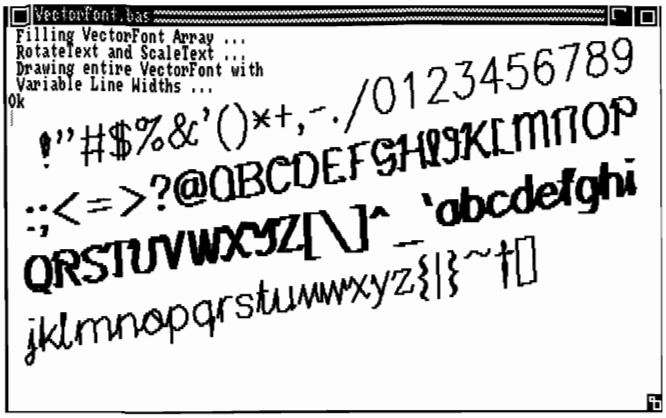
VectorFonts

The data for each character now comprises the proportional 'leftEdge' and 'rightEdge' x-relative coordinates, followed by as many xy pairs as needed to draw the character shape. The special instruction to 'move to', rather than 'draw to', affects only the next xy pair in the sequence. The last xy pair is always the special instruction to end drawing. This data format uses bytes in pairs (for conveniently stored whole words) but it isn't the only format we could have used. For instance, each byte could have been split into two parts, a direction and a distance, for a form of polar coordinates. Or, limiting each byte to a range of 63 to -64, the high bit in each byte could signal that this byte is a special instruction. There are many possible formats to choose from.

Before putting all of this irregular data together to make up what I have dubbed a 'VectorFont', we have to have some idea of the means or method to access this data. Planning foresight in the design of our format eliminates trouble ahead. We could now set up an array (in M2) to hold the offsets to the start of each character's data. But arrays are not an efficient way to contain irregular data, and a startup routine needed to read through all of this data and calculate and store offsets before we even start drawing any characters would be very time consuming. Besides, M2 doesn't manipulate individual bytes very well (BYTE is an undefined type). This sounds like an easy job for an assembler, though, working at the byte level and generating the necessary offsets.

Assembling VectorFonts

Assembled source text (see *VectorFont.asm,* which contains the same font data included in Listing 3; it can be found on the disk for this issue) produced the 'fontBase' for use with the M2 routines presented here. In addition to the font offsets and the font data, a short assembly routine, fontOffset(), speeds access to the data, via the pre-calculated offsets.



Output of BASIC demo program, showing the entire vector font, scaled and rotated

This routine returns the PC-relative address of the start of a specified character's data, given the character number. Note that since our 'fontData' begins with character number 32, we first subtract 32 to find the index to the offset.

Techniques from the previous article *Interfacing with Assembly* in this series are used again here to integrate M2AMIGA with the assembly routine, and show how to gain access to a fairly large database written in assembly language. A side note: Registers D0 and D1 are often used to return parameters as previously discussed, but in cases where these same registers are also used to pass parameters, a potential conflict could exist – a case of a return value in D0, say, (from a local function procedure) clobbering the passed parameter's value also in D0. To ensure that this doesn't happen, the M2AMIGA compiler always buffers parameters designated for D0 and D1 on the stack during parameter evaluation. After that, the parameters are moved into their registers, so there is no conflict.

Transformations

Definition module *VectorFont.def* and implementation module *VectorFont.imp* form the basis for an M2AMIGA library module. The assembled fontOffset() routine and all of the word offsets and byte data were transferred from an assembly source listing and packed into words for use with INLINE().

The drawing, scaling and rotation of VectorFont characters has been split up into a number of simple transformation algorithms. Setting up ScaleText() and RotateText() simply means storing those factors as global variables so the main routine, DrawText(), can use them in its FFP (Fast Floating Point) calculations. Inserting a little assembly language into the subroutine ReadXYpair() makes for a shorter and faster byte-reading routine as well as simplifying conversion of a signed byte to a signed integer, something not directly possible in M2 (only subrange and enumeration types are sign extended). All calculations are then done with integers.

Experimenting with drawing large letters, especially on an interlace screen, quickly shows up the need for thicker line seg-

ments. In fact, the larger the letters, the thicker the line segments needed. Unfortunately, this also slows down the speed at which text is drawn. The LWidthText() routine allows for setting thicker line segements. Line segments are made thick by drawing lines parallel on either side of the main line segment to produce a uniform solid without distorting or shifting the shape of the letter. (Note the use of RectFill() here would be even slower because of the need to also sort minimum and maximum xy points.) LWidthText() requires an odd number value for a line width and draws at least 1 line segment. The actual line width drawn is n / 2 + 1. This is a rough algorithm with room for further experimentation!

The demo program (see *vfdemo.mod* in Listing 1) is a short example featuring all of the text drawing routines in action. Note that the same Amiga graphics functions Set APen(), SetDrMd(), etc. for modifying pens, modes and patterns also apply when using these VectorFont routines.

Finally, the font data

VectorFont.imp contains all the data for a proportional script-style or handwritten style font of my own crude design, comprising all ASCII alphanumeric characters, from 20H (hex), a space, to 80H (hex), an empty box. These were sketched using grid paper and individual character 'boxes', where x = 0 is the centreline and y = 0 is the baseline.

I have written a simple AmigaBasic program (*VectorFont.bas* on this issue's disk) which draws all of the characters in the font so that you can see exactly how they fit together on screen and experiment with the font data. An editor for VectorFonts would be a nice luxury. (It could even be written in AmigaBasic by some enterprising programmer out there. Hint, hint.)

There are many refinements and changes that could be made to these basic text drawing algorithms in order to get the most speed out of them: more flexibility by adding new special instructions, or a better look to the characters. By all means experiment further!

Listing 1: *vfdemo.mod*, font routine demo program

```
MODULE vfdemo;  (* Testing the VectorFont routines *)

(* M2Amiga demo program written by Anthony Bryant *)

FROM SYSTEM IMPORT
 ADDRESS,ADR;

FROM Arts IMPORT
 Assert, TermProcedure;

FROM Exec IMPORT
 GetMsg, ReplyMsg, WaitPort;

FROM Graphics IMPORT
 RastPortPtr, SetAPen, SetDrMd, jam1;

FROM Intuition IMPORT
 IntuiMessage, IntuiMessagePtr, IDCMPFlags, IDCMPFlagSet, NewWindow,
 WindowFlags, WindowFlagSet, ScreenFlags, ScreenFlagSet, Window, WindowPtr,
 CloseWindow, OpenWindow;

FROM VectorFont IMPORT
 ScaleText, RotateText, LWidthText, DrawText;


(* OpenNewWindow - prepare a standard window to display the text
 *)
PROCEDURE OpenNewWindow(): WindowPtr;
 VAR
   wdef: NewWindow; (* temporary structure *)
 BEGIN
  WITH wdef DO
   leftEdge:= 0; topEdge:= 0; width:= 480; height:= 200;
   detailPen:= 0; blockPen:= 1;
   idcmpFlags:= IDCMPFlagSet{closeWindow};
   flags:= WindowFlagSet{windowDrag, windowDepth, windowClose,
                         activate, noCareRefresh};
   firstGadget:= NIL;
   title:= ADR("VectorFont Demo");
   screen:= NIL; bitMap:= NIL;
   type:= ScreenFlagSet{wbenchScreen};
  END;
  RETURN OpenWindow(wdef);
 END OpenNewWindow;


VAR
   wp: WindowPtr;
   wrp: RastPortPtr;
   msg: IntuiMessagePtr;
   class: IDCMPFlagSet;

PROCEDURE Cleanup();  (* this will be our exit routine *)
 BEGIN
  IF (wp # NIL) THEN  CloseWindow(wp); END;
 END Cleanup;

 BEGIN
   TermProcedure(Cleanup);  (* leave a tidy Amiga *)

   wp:= WindowPtr(OpenNewWindow());
   Assert(wp # NIL, ADR("Can't open window!"));  (* oops, exit with error *)
   wrp:= wp^.rPort;

   SetAPen(wrp,1); SetDrMd(wrp,jam1);

   ScaleText(1.3,1.1);  (* set xscale and yscale *)
   RotateText(0.0);     (* zero angle in degrees *)
   LWidthText(7);       (* make it a thick font  *)
   DrawText(wrp,ADR("Vector-Scaled Fonts"),24,40);

   ScaleText(1.4,1.8);  (* re-adjust the scale factors *)
   LWidthText(3);       (* a medium thick font *)
   DrawText(wrp,ADR("{%&@ABCDFG$XYZ}"),30,90);

   ScaleText(1.1,0.8);  (* play with the scale factors, again *)
   LWidthText(5);       (* increase font thickness *)
   RotateText(+5.0);    (* skew text to a positive angle *)
   DrawText(wrp,ADR("Transactor for the Amiga"),30,160);
   RotateText(-5.0);    (* skew text to a negative angle *)
   DrawText(wrp,ADR("for the Amiga"),220,160);

   LOOP   (* now sit and wait for closeWindow message *)
      WaitPort(wp^.userPort);
      msg:= IntuiMessagePtr(GetMsg(wp^.userPort));
      IF (msg # NIL) THEN
        class:= msg^.class;
        ReplyMsg(ADDRESS(msg));
        IF (closeWindow IN class) THEN EXIT; END;
      END;
   END; (* LOOP *)
   (* TermProcedure() kicks in here *)

END vfdemo.mod
```

Listing 2: *VectorFont.def*

```
DEFINITION MODULE VectorFont;  (* VectorFont with Scaling and Rotation *)
                               (* source in M2Amiga by Anthony Bryant  *)

FROM SYSTEM IMPORT
 ADDRESS, FFP;     (* Fast Floating Point *)

FROM Graphics IMPORT
 RastPortPtr;


(* ScaleText() - set up scale factors to multiply by vector bytes
 *)
PROCEDURE ScaleText(sx, sy: FFP);


(* RotateText() - set up rotation angle to skew text string
 *)
PROCEDURE RotateText(angle: FFP);  (* angle in 1 degrees *)


(* LWidthText() - set up line width for drawn text
 *)
PROCEDURE LWidthText(lineWidth: INTEGER);


(* DrawText() - draw text string as line segments, with scaling and rotation
 *)
PROCEDURE DrawText(rp: RastPortPtr;
                   text: ADDRESS;       (* ptr to null-terminated string *)
                   xo, yo: INTEGER);   (* set left edge, baseline pixels *)

END VectorFont.def
```

Listing 3: *VectorFont.imp*

```
IMPLEMENTATION MODULE VectorFont;  (* VectorFont with Scaling and Rotation *)
                                   (* source in M2Amiga by Anthony Bryant  *)

FROM SYSTEM IMPORT
 ADDRESS, ADR, INLINE, FFP,  (* Fast Floating Point *)
 SETREG, REG;

FROM Graphics IMPORT
 RastPortPtr, Move, Draw;

FROM MathTrans IMPORT
 Sincos;  (* FFP function *)


TYPE
 CHARPtr = POINTER TO CHAR;  (* need to operate on CHARs *)


PROCEDURE fontOffset(charnum{0}: CARDINAL): LONGINT; FORWARD;

VAR  (* global variables *)
 scX,  scY: FFP;  (* scale factors *)
 sinA,cosA: FFP;  (* trig factors *)
 lnWidth: INTEGER;(* line width *)


(* ScaleText() - set up scale factors to multiply by vector bytes
 *)
PROCEDURE ScaleText(sx, sy: FFP);
 BEGIN
  scX:= sx; scY:= sy;  (* store scale factors *)
 END ScaleText;

(* RotateText() - set up rotation angle to skew text string
 *)
PROCEDURE RotateText(angle: FFP);  (* angle in 1 degrees *)
 CONST rad = 0.0174533;  (* pi/180 *)
 BEGIN
  sinA:= Sincos(angle * rad, cosA);  (* store trig factors *)
 END RotateText;


(* LWidthText() - set up line width for drawn text
 *)
PROCEDURE LWidthText(lineWidth: INTEGER);
 BEGIN
  lnWidth:= lineWidth DIV 2;  (* store line width *)
 END LWidthText;


(* DrawText() - draw text string as line segments, with scaling and rotation
 *)
PROCEDURE DrawText(rp: RastPortPtr;
                   text: ADDRESS;       (* ptr to null-terminated string *)
                   xo, yo: INTEGER);   (* set left edge, baseline pixels *)

 CONST  nul = 0C; sp = 40C; del = 177C; emptybox = 200C; (* octal! *)

 VAR  (* local variables *)
  penup: BOOLEAN;
  x,y,xp,yp,xq,yq: INTEGER;
  lastx,xoffset,xline: INTEGER;
  sx,sy,dx,dy: FFP;
  scan: CHARPtr;
  char: CHAR;
  fontAddr: ADDRESS;

 PROCEDURE ReadXYpair();  (* subroutine *)
  BEGIN
   SETREG(8,fontAddr);    (*  MOVE.L  8(A5), A3    ; ptr to local vars  *)
                          (*  MOVE.L  -46(A3), A0  ; get fontAddr to A0 *)
   INLINE(
          1018H,          (*  MOVE.B  (A0)+, D0    ; get first byte     *)
          4880H,          (*  EXT.W   D0           ; sign extend to int *)
          1218H,          (*  MOVE.B  (A0)+, D1    ; get next byte      *)
          4881H           (*  EXT.W   D1           ; sign extend to int *)
         );
   x:= INTEGER(REG(0));   (*  MOVE.W  D0, -4(A3)   ; store local var x  *)
   y:= INTEGER(REG(1));   (*  MOVE.W  D1, -6(A3)   ; store local var y  *)
   fontAddr:= REG(8);     (*  MOVE.L  A0, -46(A3)  ; update fontAddr    *)

  END ReadXYpair;

 PROCEDURE DrawChar();    (* subroutine *)
  BEGIN
   fontAddr:= fontOffset(ORD(char));  (* get start of char data *)
   ReadXYpair();    (* read bytes as leftEdge and rightEdge *)
   xoffset:= lastx - x;  (* reset to leftEdge of char *)
   lastx:= xoffset + y;  (* reset to rightEdge of char *)
   penup:= TRUE;
   LOOP  (* read bytes until y = end char *)
    ReadXYpair();  (* read bytes as  xy element pairs *)
    (* now check x for special element *)
    IF (x = -128) THEN
       (* check y for special elements *)
       IF (y = 0) THEN EXIT; END;
       IF (y = 1) THEN penup:= TRUE; END;
    ELSE  (* do xy points *)
       sx:= scX * FFP(x + xoffset);  (* scaling *)
       sy:= scY * FFP(y);
       dx:= sx * cosA + sy * sinA;   (* rotation *)
       dy:= sy * cosA - sx * sinA;
       IF (penup = TRUE) THEN
          (* just a move *)
          xp:= xo + INTEGER(dx);  yp:= yo + INTEGER(dy);
          penup:= FALSE;  (* prepare next time to draw *)
```

```
        ELSE
           (* draw lnWidth line segments *)
           xline:= lnWidth;
           xq:= xo + INTEGER(dx);  yq:= yo + INTEGER(dy);
           Move(rp,xp,yp);   Draw(rp,xq,yq);  (* at least 1 lnWidth *)
           WHILE (xline > 0) DO  (* make thick lines *)
               Move(rp,xp-xline,yp); Draw(rp,xq-xline,yq);
               Move(rp,xp+xline,yp); Draw(rp,xq+xline,yq);
                DEC(xline);
           END;
           xp:= xq;  yp:= yq;  (* do next point *)
        END;
     END;
   END;  (* LOOP *)
  END DrawChar;

 BEGIN  (* main DrawText *)
  lastx:= 0;   xoffset:= 0;
  scan:= CHARPtr(text);
  char:= scan^;
  WHILE (char # nul) DO
    IF (char < sp) OR (char > del) THEN char:= emptybox; END;
    DrawChar();  (* does the grunt work *)
    INC(scan);
    char:= scan^;
  END;
 END DrawText;

(* ---------------- the VectorFont routine and data -------------------- *)

PROCEDURE fontOffset(charnum{0}: CARDINAL): LONGINT;
 (* $E- no entry/exit code *)
BEGIN INLINE(
 (* fontOffset:     returns address of start of char data;  charnum in D0  *)
       0440H,0020H, (*  SUBI.W  #32, D0     ; subt "loChar"          *)
       48C0H,          (*  EXT.L   D0          ; use charnum as index  *)
      0D080H,          (*  ADD.L   D0, D0      ; mult by 2             *)
       41FAH,000CH,    (*  LEA     12(PC),A0   ; find fontBase         *)
       2248H,          (*  MOVE.L  A0, A1      ; save fontBase address *)
      0D1C0H,          (*  ADDA.L  D0, A0      ; add index  to base    *)
      0D2D0H,          (*  ADDA.W  (A0), A1    ; add offset WORD       *)
       2009H,          (*  MOVE.L  A1, D0      ; return address in D0  *)
       4E75H,          (*  RTS                 ; back to M2            *)

 (* fontBase:  start of the font offset WORDs *)
       00C2H,00C6H,00E6H,010CH,0126H,015CH,019EH,01E6H,01FAH,
       0212H,022AH,023EH,024CH,0260H,0268H,0276H,027EH,02A4H,
       02B0H,02D0H,02F2H,02FEH,0324H,0356H,0360H,039EH,03D0H,
       03EAH,040AH,0414H,0422H,042CH,0458H,04ACH,04D4H,0508H,
       0534H,0558H,0574H,058EH,05BEH,05DAH,05F4H,0610H,062CH,
       063CH,0664H,067CH,06AAH,06CAH,0700H,0724H,0754H,0766H,
       0786H,0794H,07AAH,07C0H,07DEH,07EEH,07FAH,0802H,080EH,
       0820H,0828H,0830H,0858H,0880H,08A6H,08D2H,08FAH,0910H,
       0940H,095CH,0974H,0996H,09AEH,09C4H,09F0H,0A0CH,0A36H,
       0A5EH,0A86H,0A9CH,0AC2H,0ADAH,0AF6H,0B06H,0B1CH,0B2EH,
       0B46H,0B5EH,0BB0H,0BB8H,0C0AH,0C22H,0C54H,

 (* fontData:  start of the font data - BYTEs packed as WORDs *)
      0FA06H,8000H,
      0FB05H,0FFEBH,0FEEDH,0FDEFH,0FFF9H, 00EBH, 02EDH, 03EFH,
       01F9H, 8001H, 00FEH,0FFFFH, 0000H, 01FFH, 00FEH, 8000H,
      0F60AH,0FDECH,0FCEDH,0FBECH,0FCEBH,0FDECH,0FDEEH,0FCF0H,
      0FBF2H, 8001H, 05ECH, 04EDH, 03ECH, 04EBH, 05ECH, 05EEH,
       04F0H, 03F2H, 8000H,
      0F60BH,0FDEBH,0FB02H, 8001H, 07EBH, 0502H, 8001H,0F9F2H,
       09F2H, 8001H,0F8FBH, 08FBH, 8000H,
      0F60AH,0FEEBH,0FE02H, 8001H, 02EBH, 0202H, 8001H, 07F0H,
       05EEH, 02EDH,0FBEEH,0F9F0H,0F9F1H,0FAF3H,0FBF4H,0FDF5H,
       03F7H, 05F8H, 06F9H, 07FBH, 07FDH, 05FFH, 0200H,0FE00H,
      0FBFFH,0F9FDH, 8000H,
      0F40CH, 09EBH,0F700H, 8001H,0FCEBH,0FEEDH,0FEEFH,0FDF1H,
      0FBF2H,0F9F2H,0F7F0H,0F7EEH,0F8EBH,0FAEBH,0FCEBH,0FEECH,
       01EDH, 04EDH, 07ECH, 09EBH, 8001H, 05F9H, 03FAH, 02FCH,
       02FEH, 0400H, 0600H, 08FFH, 09FDH, 09FBH, 07F9H, 05F9H,
       8000H,
      0F30DH, 0AF4H, 0AF3H, 09F2H, 08F2H, 07F3H, 06F5H, 04FAH,
       02FDH, 00FFH,0FE00H,0FA00H,0F8FFH,0F7FEH,0F6FCH,0F6FAH,
      0F7F8H,0F8F7H,0FFF3H, 00F2H, 01F0H, 01EEH, 00ECH,0FEEBH,
      0FCECH,0FBEEH,0FBF0H,0FCF3H,0FEF6H, 03FDH, 05FFH, 0700H,
       0900H, 0AFFH, 0AFEH, 8000H,
      0FB05H, 01ECH, 00EDH,0FFECH, 00EBH, 01ECH, 01EEH, 00F0H,
      0FFF2H, 8000H,
      0F907H, 04EBH, 02EDH, 00EFH,0FEF2H,0FDF7H,0FDFBH,0FE00H,
       0003H, 0205H, 0407H, 8000H,
      0F907H,0FCEBH,0FEEDH, 00EFH, 02F2H, 03F7H, 03FBH, 0200H,
       0003H,0FE05H,0FC07H, 8000H,
      0F808H,0FBF7H, 05F7H, 8001H,0FBF2H, 05FCH, 8001H, 05F2H,
      0FBFCH, 8000H,
      0F808H, 00F1H, 00FDH, 8001H,0FBF7H, 05F7H, 8000H,
      0FB05H, 01FFH, 0000H,0FFFFH, 00FEH, 01FFH, 0101H, 0003H,
      0FF04H, 8000H,
      0F808H,0FBF7H, 05F7H, 8000H,
      0FB05H, 00FEH,0FFFFH, 0000H, 01FFH, 00FEH, 8000H,
      0F50BH, 09EBH,0F707H, 8000H,
      0F60AH,0FFEBH,0FCECH,0FAEFH,0F9F4H,0F9F7H,0FAFCH,0FCFFH,
      0FF00H, 0100H, 04FFH, 06FCH, 07F7H, 07F4H, 06EFH, 04ECH,
       01EBH,0FFEBH, 8000H,
      0F60AH,0FCEFH,0FEEEH, 01EBH, 0100H, 8000H,
      0F60AH,0FAF0H,0FAEFH,0FBEDH,0FCECH,0FEEBH, 02EBH, 04ECH,
       05EDH, 06EFH, 06F0H, 05F3H, 03F6H,0F900H, 0700H, 8000H,
      0F60AH,0FBEBH, 06EBH, 00F3H, 03F3H, 05F4H, 06F5H, 07F8H,
       07FAH, 06FDH, 04FFH, 0100H,0FE00H,0FBFFH,0FAFEH,0F9FCH,
       8000H,
      0F60AH, 0300H, 03EBH,0F9F9H, 08F9H, 8000H,
      0F60AH, 05EBH,0FBEBH,0FAF4H,0FBF3H,0FEF2H, 01F2H, 04F3H,
       06F5H, 07F8H, 07FAH, 06FDH, 04FFH, 0100H,0FE00H,0FBFFH,
      0FAFEH,0F9FCH, 8000H,
      0F60AH, 06EEH, 05ECH, 02EBH, 00EBH,0FDECH,0FBEFH,0FAF4H,
      0FAF9H,0FBFDH,0FDFFH, 0000H, 0100H, 04FFH, 06FDH, 07FAH,
       07F9H, 06F6H, 04F4H, 01F3H, 00F3H,0FDF4H,0FBF6H,0FAF9H,
       8000H,
      0F60AH,0F9EBH, 07EBH,0FD00H, 8000H,
      0F60AH,0FEEBH,0FBECH,0FAEEH,0FAF0H,0FBF2H,0FDF3H, 01F4H,
       04F5H, 06F7H, 07F9H, 07FCH, 06FEH, 05FFH, 0200H,0FE00H,
      0FBFFH,0FAFEH,0F9FCH,0F9F9H,0FAF7H,0FCF5H,0FFF4H, 03F3H,
       05F2H, 06F0H, 06EEH, 05ECH, 02EBH,0FEEBH, 8000H,
      0F60AH, 06F2H, 05F5H, 03F7H, 00F8H,0FFF8H,0FCF7H,0FAF5H,
      0F9F2H,0F9F1H,0FAEEH,0FCECH,0FFEBH, 00EBH, 03ECH, 05EEH,
       06F2H, 06F7H, 05FCH, 03FFH, 0000H,0FE00H,0FBFFH,0FAFDH,
       8000H,
      0FB05H, 00F2H,0FFF3H, 00F4H, 01F3H, 00F2H, 8001H, 00FEH,
      0FFFFH, 0000H, 01FFH, 00FEH, 8000H,
      0FB05H, 00F2H,0FFF3H, 00F4H, 01F3H, 00F2H, 8001H, 01FFH,
       0000H,0FFFFH, 00FEH, 01FFH, 0101H, 0003H,0FF04H, 8000H,
      0F40CH, 08EEH,0F8F7H, 0800H, 8000H,
      0F30DH,0F9F4H, 07F4H, 8001H,0F9FAH, 07FAH, 8000H,
      0F40CH,0F8EEH, 08F7H,0F800H, 8000H,
      0F709H,0FAF0H,0FAEFH,0FBEDH,0FCECH,0FEEBH, 02EBH, 04ECH,
       05EDH, 06EFH, 06F1H, 05F3H, 04F4H, 00F6H, 00F9H, 8001H,
       00FEH,0FFFFH, 0000H, 01FFH, 00FEH, 8000H,
      0F30EH, 05F3H, 04F1H, 02F0H,0FFF0H,0FDF1H,0FCF2H,0FBF5H,
      0FBF8H,0FCFAH,0FEFBH, 01FBH, 03FAH, 04F8H, 05F0H, 04F8H,
       06FBH, 08FBH, 0AFBH, 0BF6H, 0BF4H, 0AEFH, 09EDH, 07ECH,
       05ECH, 03EBH,0FFEBH,0FCECH,0FAEDH,0F8EFH,0F7F1H,0F6F4H,
      0F6F7H,0F7FAH,0F8FCH,0FAFEH,0FCFFH,0FF00H, 0200H, 06FFH,
       0AFEH, 8000H,
      0F60BH, 07F0H, 06EEH, 04ECH, 02EBH,0FEEBH,0FBEEH,0FAF0H,
      0F9F3H,0F9FBH,0FAFDH,0FDFFH,0FF00H, 0300H, 05FFH, 07FDH,
       07F0H, 0800H, 09FFH, 8000H,
      0F50AH,0F6FFH,0F900H,0F9EBH,0F6ECH, 02EBH, 05ECH, 06EDH,
       07EFH, 07F1H, 06F3H, 05F4H, 02F5H,0F9F5H, 02F5H, 05F6H,
       06F7H, 07F9H, 08FBH, 08FDH, 08FEH, 07FFH, 05FFH, 0200H,
      0F900H, 8000H,
      0F60BH, 01EEH, 04F0H, 08F0H, 07EEH, 05ECH, 03EBH,0FFEBH,
      0FDECH,0FBEEH,0FAF0H,0F9F3H,0F9F8H,0FAFBH,0FBFDH,0FDFFH,
      0FF00H, 0300H, 05FFH, 07FDH, 08FBH, 8000H,
```

```
0F50AH,0F6EFH,0F9EBH, 00EBH, 03ECH, 05EEH, 06F0H, 07F3H,
 07F8H, 06FBH, 05FDH, 03FFH, 0000H,0F9FEH,0F8FDH,0F8F3H,
0F9EEH, 8000H,0F609H, 05EDH, 03EBH,0FBEBH,0FAEDH,0FAF5H,
 02F5H,0FAF4H,0FAFEH,0FB00H, 0600H, 07FFH, 05FDH, 8000H,
0F608H,0FAEDH,0FCEBH, 07EBH, 8001H,0FCEBH,0FCF5H, 02F5H,
0FCF4H,0FCFEH,0FB00H,0FAFEH, 8000H,
0F60BH, 01EEH, 04F0H, 08F0H, 07EEH, 05ECH, 03EBH,0FEEBH,
0FCECH,0FBEDH,0FAEEH,0F9F3H,0FDF9H, 05F7H, 06F5H, 08F5H,
 08FBH, 07FDH, 05FFH, 0300H,0FF00H,0FDFFH,0FDFDH, 8000H,
0F50BH,0F6EFH,0F9EBH,0FBEBH,0F900H,0F6FEH, 8001H, 08EBH,
 06EDH, 0800H, 8001H,0F9F5H, 07F5H, 8000H,
0FC06H,0FCFDH,0FEFFH, 0000H, 04F2H, 04EDH, 02EBH, 00EBH,
0FEEDH,0FEF1H, 02FEH, 0400H, 8000H,0F808H, 06F5H,0FEF2H,
0FCEDH,0FEEBH, 04EDH, 04FBH, 03FEH, 02FFH, 0000H,0FE00H,
0FCFFH,0FBFEH, 8000H,
0F50AH,0F6EFH,0F9EBH,0FBEBH,0F900H,0F6FEH, 8001H, 07EBH,
0F9F9H, 8001H,0FEF4H, 0700H, 09FEH, 8000H,
0F607H, 00EDH,0FEEBH,0FAEBH,0FA00H,0FAFEH, 0600H, 8000H,
0F10FH,0F3EFH,0F5EBH,0F500H, 8001H,0F5EFH,0F8ECH,0FAEBH,
0FDEBH,0FFECH, 00EFH, 0000H, 8001H, 00EFH, 03ECH, 05EBH,
 08EBH, 0AEFH, 0B00H, 8000H,
0F50BH,0F6F0H,0F9ECH,0F900H, 8001H,0F9EEH,0FBEBH, 07EBH,
 07EEH, 05F0H, 0500H, 8000H,
0F50BH,0FEEBH,0FCECH,0FAEEH,0F9F0H,0F8F3H,0F8F8H,0F9FBH,
0FAFDH,0FCFFH,0FE00H, 0200H, 04FFH, 06FDH, 07FBH, 08F8H,
 08F3H, 07F0H, 06EEH, 04ECH, 02EBH,0FEEBH, 8000H,
0F50AH,0F6EFH,0F9EBH, 02EBH, 05ECH, 06EDH, 07EFH, 07F3H,
 06F5H, 05F5H, 03F6H,0F9EFH,0FAEDH,0F900H,0F7FEH, 8000H,
0F50BH,0FEEBH,0FCECH,0FAEEH,0F9F0H,0F8F3H,0F8F8H,0F9FBH,
0FAFDH,0FCFFH,0FE00H, 0200H, 04FFH, 06FDH, 07FBH, 08F8H,
 08F3H, 07F0H, 06EEH, 04ECH, 02EBH,0FEEBH, 8001H,0FAFDH,
 01FCH, 0702H, 8000H,
0F50AH,0F6EFH,0F9EBH, 02EBH, 05ECH, 06EDH, 07EFH, 07F3H,
 06F5H, 05F6H, 02F5H,0F9F0H,0F900H,0F7FEH, 8001H, 00F5H,
 0700H, 8000H,
0F60AH, 05EFH, 07EEH, 05ECH, 02EBH,0FEEBH,0FBECH,0F9EEH,
0F9F0H,0FAF2H,0FBF3H,0FDF4H, 03F6H, 05F7H, 06F8H, 07FAH,
 07FDH, 05FFH, 0200H,0FE00H,0FBFFH,0F9FDH,0FBFDH, 8000H,
0F808H,0F6EFH,0F9EBH, 07EBH, 8001H, 00EBH, 0000H,0FEFEH,
 8000H,
0F50BH,0F6EFH,0F9EBH,0FBEDH,0FBF0H,0F9FAH,0FAFDH,0FCFFH,
0FF00H, 0100H, 04FFH, 06FDH, 07FAH, 07EDH, 09EBH, 8000H,
0F50BH,0F6EFH,0F9EBH,0FBEDH, 0000H, 07EBH, 8000H,
0F30DH,0F6EBH,0F9FEH,0FB00H,0FDFEH, 00EBH, 03FEH, 0500H,
 07FEH, 0AEBH, 8000H,
0F60AH,0F7EFH,0F9EBH, 0700H, 09FEH, 8001H, 09EFH, 07EBH,
0F900H,0F7FEH, 8000H,
0F60AH,0F7EFH,0F9EBH,0FBEDH,0FDF5H, 00F5H, 07EBH, 08FBH,
 07FDH, 05FFH, 0300H,0FF00H,0FDFFH,0FDFDH, 8000H,
0F60AH,0F7EFH,0F9EBH, 07EBH,0F900H, 0700H, 09FEH, 8000H,
0F907H, 04EBH,0FDEBH,0FD07H, 0407H, 8000H,
0F50BH,0F7EBH, 0907H, 8000H,
0F907H,0FCEBH, 03EBH, 0307H,0FC07H, 8000H,
0F808H,0FEF1H, 00EEH, 02F1H, 8001H,0FBF4H, 00EFH, 05F4H,
 8000H,
0F30DH,0F707H, 0907H, 8000H,
0FC04H,0FDEBH, 00F2H, 8000H,
0F70AH, 06F5H, 04F3H, 02F2H,0FFF2H,0FDF3H,0FBF5H,0FAF8H,
0FAFAH,0FBFDH,0FDFFH,0FF00H, 0200H, 04FFH, 06FDH, 8001H,
 06F2H, 0600H, 08FDH, 8000H,
0F609H,0F7FDH,0FAEBH,0FA00H, 8001H,0FAF5H,0FCF3H,0FEF2H,
 01F2H, 03F3H, 05F5H, 06F8H, 06FAH, 05FDH, 03FFH, 0100H,
0FE00H,0FCFFH,0FAFDH, 8000H,
0F709H,0F7FDH,0FAFAH,0FAF8H,0FBF5H,0FDF3H,0FFF2H, 02F2H,
 04F3H, 06F5H, 8001H,0FAFAH,0FBFDH,0FDFFH,0FF00H, 0200H,
 04FFH, 06FDH, 8000H,
0F70AH,0F7FDH,0FAFAH,0FAF8H,0FBF5H,0FDF3H,0FFF2H, 02F2H,
 04F3H, 06F5H, 06EBH, 0600H, 08FDH, 8001H,0FAFAH,0FBFDH,
0FDFFH,0FF00H, 0200H, 04FFH, 06FDH, 8000H,
0F709H,0F7FDH,0FAF8H, 06F8H, 06F6H, 05F4H, 04F3H, 02F2H,
0FFF2H,0FDF3H,0FBF5H,0FAF8H,0FAFAH,0FBFDH,0FDFFH,0FF00H,
 0200H, 05FFH, 08FDH, 8000H,
0FB07H,0FCFDH, 06EBH, 03EBH, 01ECH, 01EFH, 0100H, 8001H,
0FDF2H, 04F2H, 8000H,
0F70AH, 06F2H, 0602H, 0505H, 0406H, 0207H,0FE07H,0FC06H,
 8001H, 06F5H, 04F3H, 02F2H,0FFF2H,0FDF3H,0FBF5H,0FAF8H,
0FAFAH,0FBFDH,0FDFFH,0FF00H, 0200H, 04FFH, 06FDH, 8000H,
0F70AH,0F7FDH,0FBEBH,0FB00H, 8001H,0FBF6H,0FEF3H, 00F2H,
 03F2H, 05F3H, 06F6H, 0600H, 08FDH, 8000H,
0FC04H,0FFEBH, 00ECH, 01EBH, 00EBH,0FFEBH, 8001H,0FEFDH,
 00F2H, 0000H, 02FDH, 8000H,
0FB05H, 00EBH, 01ECH, 02EBH, 01EBH, 00EBH, 8001H,0FEFDH,
 01F2H, 0103H, 0006H,0FE07H,0FD07H,0FC05H,0FC03H, 04FDH,
 8000H,
0F708H,0F9FDH,0FBEBH,0FB00H, 8001H, 05F2H,0FBFCH, 8001H,
0FFF8H, 0600H, 08FDH, 8000H,
0FE09H,0FEFDH, 02EDH, 02EBH, 00EBH, 00FCH, 01FFH, 0300H,
 0500H, 06FDH, 8000H,
0F10FH,0F2FDH,0F5F2H,0F500H, 8001H,0F5F6H,0F8F3H,0FAF2H,
0FDF2H,0FFF3H, 00F6H, 0000H, 8001H, 00F6H, 03F3H, 05F2H,
 08F2H, 0AF3H, 0BF6H, 0B00H, 0EFDH, 8000H,
0F70AH,0F8FDH,0FBF2H,0FB00H, 8001H,0FBF6H,0FEF3H, 00F2H,
 03F2H, 05F3H, 06F6H, 0600H, 08FDH, 8000H,
0F70AH,0F7FDH,0FFF2H,0FDF3H,0FBF5H,0FAF8H,0FAFAH,0FBFDH,
0FDFFH,0FF00H, 0200H, 04FFH, 06FDH, 07FAH, 07F8H, 06F5H,
 04F3H, 02F2H,0FFF2H, 08FDH, 8000H,
0F609H,0F7FDH,0FAF2H,0FA07H, 8001H,0FAF5H,0FCF3H,0FEF2H,
 01F2H, 03F3H, 05F5H, 06F8H, 06FAH, 05FDH, 03FFH, 0100H,
0FE00H,0FCFFH,0FAFDH, 8000H,
0F70AH, 06F2H, 0607H, 08FDH, 8001H, 06F5H, 04F3H, 02F2H,
0FFF2H,0FDF3H,0FBF5H,0FAF8H,0FAFAH,0FBFDH,0FDFFH,0FF00H,
 0200H, 04FFH, 06FDH, 8000H,
0F906H,0FAFDH,0FDF2H,0FD00H, 8001H,0FDF8H,0FEF5H, 00F3H,
 02F2H, 05F2H, 8000H,
0F809H, 06F5H, 05F3H, 02F2H,0FFF2H,0FCF3H,0FBF5H,0FCF7H,
0FEF8H, 03F9H, 05FAH, 06FCH, 06FDH, 05FFH, 0200H,0FF00H,
0FCFFH,0FBFDH, 8000H,
0FB07H,0FCFDH, 00EBH, 00FCH, 01FFH, 0300H, 0500H, 08FDH,
 8001H,0FDF2H, 04F2H, 8000H,
0F70AH,0F8FDH,0FBF2H,0FBFCH,0FCFFH,0FE00H, 0100H, 03FFH,
 06FCH, 8001H, 06F2H, 0600H, 08FDH, 8000H,
0F808H,0F9FDH,0FDF2H, 0000H, 06F2H, 06F7H, 08FDH, 8000H,
0F50BH,0F7FDH,0F8F2H,0FCFFH,0FC00H, 00F2H, 04FFH, 0400H,
 08F2H, 0AF8H, 8000H,
0F809H,0F9F7H,0FBF2H, 0600H, 08FDH, 8001H, 06F2H,0FB00H,
 8000H,
0F808H,0F9F7H,0FAF2H, 0000H, 8001H, 06F2H, 0000H,0FE04H,
0FC06H,0FA07H,0F907H, 8000H,
0F809H, 06F2H,0FB00H, 8001H,0F9F7H,0FBF2H, 06F2H, 8001H,
0FB00H, 0600H, 08FDH, 8000H,
0F907H, 02EBH, 00ECH,0FFEDH,0FEEEH,0FEF0H,0FFF1H, 00F2H,
 01F4H, 01F6H,0FFF8H, 8001H, 00ECH,0FFEEH,0FFEFH, 00F0H,
 01F1H, 02F3H, 02F5H, 01F7H,0FDF9H, 01FBH, 02FDH, 02FFH,
 0101H, 0002H,0FF03H,0FF04H, 0006H, 8001H,0FFFAH, 01FCH,
 01FEH, 0000H,0FF01H,0FE02H,0FE04H,0FF05H, 0006H, 0207H,
 8000H,
0FC04H, 00EBH, 0007H, 8000H,
0F907H,0FEEBH, 00ECH, 01EDH, 02EEH, 02F0H, 01F1H, 00F2H,
0FFF4H,0FFF6H, 01F8H, 8001H, 00ECH, 01EEH, 01EFH, 00F0H,
0FFF1H,0FEF3H,0FEF5H,0FFF7H, 03F9H,0FFFBH,0FEFDH,0FEFFH,
0FF01H, 0002H, 0103H, 0104H, 0006H, 8001H, 01FAH,0FFFCH,
0FFFEH, 0000H, 0101H, 0202H, 0204H, 0105H, 0006H,0FE07H,
 8000H,
0F40CH,0F7F1H,0F8F1H,0FAF0H,0FCEEH,0FEEEH, 02F0H, 04F1H,
 06F1H, 08F0H, 09EEH, 8000H,
0F808H, 00EBH,0FFEDH, 00EFH, 01EDH, 00EBH, 8001H, 00EBH,
 0007H, 8001H, 00F6H,0FFF9H, 0007H, 01F9H, 00F6H, 8001H,
0FAF2H, 06F2H, 8001H, 02F2H, 04F3H, 06F2H, 04F1H, 02F2H,
 8000H,
0F808H,0FAEBH,0FA07H, 0607H, 06EBH,0FAEBH, 8000H
  );
END fontOffset;

END VectorFont.imp
```

ReSource

The interactive unassembler

Review by Jim Butterfield
Copyright © 1989 Jim Butterfield

ReSource, version 3, *by Glen McDiarmid*

Distributed in North America by:
The Puzzle Factory
P.O. Box 986
Veneta, Oregon 97487, U.S.A.
(503) 935-3709
Price: $75.00 (US)

In current usage, a *disassembler* is a program that displays information about a machine language program so that the user can view it. Most debugging packages will perform disassembly. An *unassembler* does more: it translates the program rigorously, so that it could be rebuilt using an assembler. *ReSource*, by Glen McDiarmid, is a fine interactive unassembler.

A demonstration version of the current *ReSource* (version 3) is circulating on networks and bulletin boards. It has all the features of the commercial release, except for one thing: it will generate no output to files or printer. In a sense, the demo is a fine disassembler: you can look, but you can't reconstruct. Earlier versions of *ReSource* have been around for some time, but version 3 is more polished and has extra features.

Why would a user want to pick apart a program's code? To see how it works, of course; perhaps also to investigate how certain effects are accomplished, or to attempt to correct a perceived operational anomaly. For my part, I like to investigate my own work: small programs written in other languages. This allows me to see the machine code that has been created by the compiler. Sometimes I think the compiler has done a fine job, and admire its efficient coding techniques. On other occasions, I feel that the resulting code can be greatly improved; *ReSource* will allow me to bring the program back to Assembly language source (to re-source, in fact) and rebuild it to my taste.

System needs

ReSource uses the ARP library, which must be in place in your LIBS: directory. This should be no hardship, since ARP is freely available, and has recently been newly released in a form highly compatible with Workbench 1.3. Interested users and user groups should write: AmigaDos Replacement Project (ARP), c/o MicroSmiths, Inc., PO Box 561, Cambridge MA, USA 02140.

ReSource needs an Amiga with at least one megabyte of RAM in order to be useful, and two megs is better. When disassembling a file, the program uses six to ten times as much memory as the file itself. For example: to disassemble a 100K file you will need abut 850K of available memory, of which 400K must be contiguous.

Huge files will need more attention in any case. Although *ReSource* performs very intelligent work on the program, you will undoubtedly want to add meaningful labels to the result. When the file is very large, this will take quite a bit of study.

Files containing overlay segments will not be successfully loaded in current versions of *ReSource*.

Basic operation

The program to be studied is loaded. *ReSource* has a huge roster of commands, made available through 14 menus. The most popular commands may be mapped to the keyboard so that a single keypress will invoke a desired action; a suggested keymap is supplied with the program.

The user will start a session with a few commands that will cause *ReSource* to scan the program, assign labels to appropriate places, automatically estimate which parts of the program contain code, data, or strings, and tidy things up for good readability.

A *ReSource* Walkthrough

ReSource is so rich in commands that it's hard for the new user to cope with them all. Using the demo or commercial version, try this session: it's intended to show you how elegantly *ReSource* can cope with unknown information.

This session will partially unassemble the AmigaDOS *Bind-Drivers* command. Before starting, be sure that the *RS.Macros* and *RS.KeyTable* files (provided with the demo or commercial version) have been copied into your s: directory. These files will make the session go much faster. Also, be sure that you have placed *arp.library* into your LIBS: drawer.

Start the session with:

```
ReSource c:BindDrivers
```

or, in the demo version:

```
RsDemo c:BindDrivers
```

There will be a pause while the macros and keymaps load, and then a disassembly of *BindDrivers* will appear on the screen. Hold down the left mouse button, and scroll the disassembly up and down to get a feel for speed and control. When you're ready, press **ALT-cursor-up** and return to the start of the document.

The first job is to ask *ReSource* to put labels into the text. Press F1 to trigger the function *Labels/Create Multiple/All* (i.e. F1 is a shortcut for the *All* submenu-item under the *Create Multiple* menu-item on the *Labels* menu; other menu selections in this walkthrough will use this kind of notation). The labels will be placed, but the display will stop at a point where *ReSource* believes it has found data rather than program code. Skip over the data by touching F2 (*Cursor/Relative/Next Unparsed Code*) and then press F1 again to generate more labels. You're not at the end yet; but another cycle through F2 and F1 will take you there, and the display will return to the beginning of the listing.

Following this first pass, press F4 (*Display/Fill-in Data Types*), and the program will tidy and 'firm up' the listing in its present state. This will make forward and backward scrolling smoother, and will build strings into readable groups.

The program is already quite readable: the strings have been given meaningful labels, for example. If the source ("BindDrivers.asm") were to be saved now, it would assemble successfully. Of course, you can't save the source if you have the demo version.

Near the top of the program, you will spot the line **move.l 4,a6**, which sets up a pointer to the Exec library in register A6. Scroll down so that the following line (**move.l a6,lbl001D8**) is at the top of the screen. This command saves the pointer into memory. Let's go to that place in memory; just press the cursor-right key. Now we're there, press 9 on the numeric pad, and type in the label **ExecBase.** Go back to where we were working by pressing cursor-left, and note that the label has been correctly changed here, too.

There's a call to Exec not far down: **jsr -$00c6(a6)**. Bring it to the top of the screen. Touch the **e** key (for *Symbols/Exec*) and you'll see the label **_LVOAllocMem** appear in place of **-$00C6**. Now we know that we're doing a memory allocation, back up a couple of instructions to **move.l #$0019c**, which will set up the number of bytes to be allocated.

Chances are you'd rather see this number in decimal. Pull down menu *D* (Display) and select item *Set Numeric Base*, sub-item *Decimal*. The value 412 replaces the earlier hexadecimal value.

The following instruction, also a **move.l**, indicates the type of memory requested. Bring this line to the top and then select *S2* (Symbols)/*M-N/Memory attributes*. The previous value of #$00010001 will change to #MEMF_PUBLIC|MEMF_CLEAR. As you look through the menus, you may begin to appreciate the huge size of the data embedded in the program.

Move along to the line with the label **lbC000032** and tap **9** on the numeric key. Type in the label **ALLOC1OK**. Move down a bit further to the line containing **jsr -$0126(a6)**; touch the letter **E** and note that this is a call to Exec *FindTask*. Repeating the procedure with **jsr -$228(a6)** below, we see an *OpenLibrary* call for the DOS library. The DOS-name string is properly labelled; *ReSource* did this for us automatically. Line up to the **bne.s** line just below, and touch the cursor-right key, followed by the numeric pad **9.** Identify this line as **DosOpen**, and then press **cursor-left** to return to your earlier working point. A few lines down, the **jsr -$006c(a6)** can be identified by pressing the **E** key; since it's a call to *Alert*, we can move back a couple of lines and translate #$00038007 by using menu *S2/A-B/Alert codes*. Guru codes might start making sense after you see how this coding is done.

Go down a few lines to **move.l d0**; recalling the above program logic, we know that D0 will hold a pointer to our DOS library. Press **cursor-right,** then **9,** and enter the label **DosBase**. Follow with **cursor-left** to come back to our main code.

The next line has a reference to **$00ac(a4)**. Looking back again, you might see that A4 contains a pointer to the program's Task block (actually, a Process structure); tap **T** for *Task* and you'll see that the reference is to *pr_CLI*, telling us whether the program has been called into existence from the CLI or from the Workbench.

We'll end our exercise at this point. Soon, this code will venture into B-strings and B-pointers, which are not hard but take a little practice to understand properly.

And if you want to browse further on your own, you'll discover a curious set of subroutines near the end of BindDrivers whose only purpose is to rearrange register usage so as to safely call various library functions. These subroutines are called 'stubs'. Modern C compilers use them less than used to be the case, since new methods ('pragmas') permit these calls to be made in-line with little chance of register mix-ups.

We've only touched a few of the commands of *ReSource*. Even so, the power of its operation is quite visible.

Another exercise may be performed in two steps. In menu *LM* (Local Macros), select *Execution Speed/Wait on Mouse*. Now go to *GM* (Global Macros) and pick one of the demonstration disassemblies. This will be performed one step at a time; each step will wait for you to click on the mouse. You'll be able to watch *ReSource* do its thing on a variety of commands.

This procedure is both quick and amazingly intuitive. (See "A *ReSource* Walkthrough" for more details on using the program.)

The program may be inspected by scrolling using the mouse or keyboard. Labels or symbols may be entered manually; in many cases, *ReSource* will supply them automatically if given a hint as to what is wanted. Thus, **jsr -$0228(a6)** can be instantly translated to **jsr _LvoOpenLibrary(A6)** if the user requests an exec-library translation. Similarly, **move.l #$003ed,d1** will be changed to **move.l #mode_oldfile,d1** by a request to identify 'file type'. The supplied names will also be defined in the resulting source as EQU or XREF statements.

Hundreds of libraries, structures, and constant definitions are held within the *ReSource* program. Virtually everything that might be found within the standard include files is there, plus extras such as ARP and ARexx definitions.

I have not found a way to add my own structure definitions. That's a pity, since other custom libraries would be useful from time to time. For example: the HiSoft Basic library would be a handy addition.

Programs may be readily commented. Both the full-line comments, and the short note at the end of a line of code are simple to do. They help greatly in developing understanding of a program's activities.

Everything is fast. The listing scrolls quickly, and various operations such as searching or finding label usage are speedy. The user can override decisions made by *ReSource*, and will need to do so with large and complex programs.

Even so, the original *ReSource* work is substantially correct.

Documentation is extensive (yes, the demo version also comes with full documentation on disk). Even so, the rich set of commands is hard to cover completely. Sometimes, the text gets a little obscure or starts to give elaborate references to other areas of the documentation. Most user questions will be resolved by just trying a menu command to see what happens.

Special file types

ReSource will investigate more than just executable program files. You can decide to look at a section of RAM memory; you can load a data file; you may even look at specific sectors on disk. This last is particularly useful if you wish to view the contents of a disk boot block.

You may bring in a file as a 'binary image' rather than in its more usual 'load' fashion. In this case, you will be allowed to change the contents of the file and then save it back. In this case, you cannot save the source; but the binary approach is good for quick program patches.

If you're picking apart a file, and need to do more work on it later, you may save it as a special format **.RS** (*ReSource*) file. With this kind of file, you may pick up the file later and continue where you left off.

Wish list

ReSource is so fully-featured and flexible that it seems almost ungracious to itemize things that I'd like the program to do for me. Indeed, it may be that the operations I hope for here already exist among the thousand or more menu commands currently implemented. In a system so rich, it's easy to miss some bits.

I'd like to be able to set a 'range' within the program under study: for example a program module or subroutine. The area between LINK and UNLK in particular is often a special area within which a user would like to pick out certain information, without accidentally slopping over into 'outside' code.

Programs within memory may be disassembled, but for some reason certain jumps within this type of program refuse to permit themselves to be labelled.

A special 'A4' mode is available for the handling of certain types of C-compiled programs. This is useful, but I'd be happy if I were able to designate the register, rather than just having to live with A4 and no other.

Where a block of memory is allocated to hold work variables, it would be nice to have a simple way of labelling the areas within such a block. At the moment, search-and-replace seems to be the only way to go about the task. This is more work than fun. A 'user-defined structure' might be a good way to approach this task.

If a place in the program has been specifically defined to be a certain type, the type may be changed at a later time: you're not stuck. But I'd like to be able to 'free' the type designation of such a location; in effect, this would be saying, 'just follow along with whatever the currently defined type is'. I don't seem to be able to do this. Once I've tried out a type - perhaps experimentally to see whether a certain area might be code or ASCII or whatever - I cannot free it to its original state. It's 'type cast'.

Summary

The program is astonishing in many ways. The massive size of its internal tables boggles the mind. I admire the remarkable accuracy with which it makes intuitive guesses at the nature of certain bytes.

ReSource is of no value to users who have no interest in assembly language. For those who do, it can be a valuable tool for both development and learning.

You could use the free demo as a learning tool, to help you view the working of machine language. Even if that's your only objective, chances are you'll be impressed with the fine workmanship of the package, and want to send off for the commercial version. Considering what it does, it's quite a bargain. ◘

FutureSound 500

A tool for programmers - and a great toy

Review by Chris Zamara

Stereo sound digitizing hardware/software

from Applied Visions, Inc., Suite 2200 One Kendall Square, Cambridge, MA 02139

Suggested Retail: $119.95 (US)

FutureSound is a sound digitizing system for the Amiga 500/2000 series. The hardware consists of a small box that plugs into the parallel port and has a subminiature microphone input jack, two RCA-plug line inputs, and a level control. You can use the dual line inputs to digitize in stereo, or a single microphone (not included) for mono input. The supplied software lets you monitor the incoming sound, view the input level on a graphic 'VU-meter', digitize sound samples, and manipulate the digitized data. A highly readable 44-page manual completes the package.

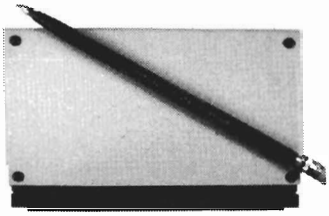
A sound digitizer might be of use in creating new 'instruments' for music programs or other end-user software, but it will probably appeal even more strongly to programmers, since it will create the data for sound samples that can be incorporated into programs. Sound effects of all kinds can be easily generated verbally, from instruments or household objects, or found on existing recordings. Digitizing sounds in this manner is much easier – and usually better – than generating sounds algorithmically. For the programmer, FutureSound is an ideal package, since it contains C and BASIC source code to play digitized sound sample files in both the standard IFF-8SVX format and the unique FutureSound format. By using the supplied code and the FutureSound program and hardware, it's a simple matter to put your own speech or any sound you like into your own programs. Since the resulting files are in IFF format, any program that lets you import 'instruments' or sound effects will be able to use these sounds. We also discovered that, as dry and scientific as 'sound sample digitization' may sound, the process of recording and later manipulating bizarre noises can be more fun than the best of video games.

The FutureSound program – used for capturing and editing the sound samples – is easy to use, but limited and a bit clumsy for some operations. It provides the necessary features for capturing sound samples, but could go a lot further in its analyzing and editing capabilities. Fortunately, there are several commercial and freely-redistributable programs that do the same job but are more extensive, so you can just use the FutureSound hardware and choose the software that best suits your needs. In our testing, we discovered an annoying bug in the program: after using it, the system slows down to a crawl while displaying a high-res four-bitplane screen; a reboot seemed to be the only solution.

The program lets you 'record' sound in up to four 'tracks'. Before recording, you can monitor the sound through the monitor's speaker and view the recording level on a meter to allow proper adjustment of the level control. You can specify the length of the recording in seconds, the sampling rate in samples per second, and the number of bytes used to store the sample – changing any of these variables affects one of the others. Once recorded, you can view the sound data on a graph, and set start and end points for the sample. You can also zoom in to different levels of magnification, and perform some rudimentary transformations on the data or any section thereof, like scaling, reversing, copying to another track, or mixing with another track.

The program's user interface is easy to use, but lacks the 'slickness' we've come to expect of Amiga software. Most importantly, however, the basic features are there and the whole process of digitizing, selecting a specific range, and saving the sound file is about as uncomplicated as one could hope for.

The sound quality of the samples we made seemed quite good, especially for speech. Good results, simple operation, a degree of control over the sampled data, and source code for use in programs lead us to recommend this package. For the programmer, FutureSound 500 is a useful tool that doubles as a giggle-a-minute toy.



MiniMegs external memory module by Memory And Storage Technology Inc.

Retail Prices: 512k $329, 1 Meg $489, 2 Meg $779.

Review by Malcolm O'Brien

If you're using a standard Amiga 500 with the A501 RAM expander you're cruising at a cool one megabyte. You probably thought that that was more than enough memory. Well, maybe it is... but maybe it isn't. While working with just such a system recently I discovered that I didn't have enough memory left to save my work under a different name!

Maybe your software requirements are not as extensive as mine... but maybe they are. If so, you should take a serious look at the MiniMegs External Memory Module from M.A.S.T. This tiny (5 1/4" by 3" by 3/4") box is a pocket-sized powerhouse that is available in two configurations: one for the A500 and one for the A1000. Both models sport two LEDs: a green Power/Config LED and a red one called a "RAM tachometer"; it lights when MiniMegs RAM is being accessed.

Installation is a snap. Just remove the plastic cover over the Amiga's 86-pin expansion bus and (with the computer turned off!) slide the MiniMegs edge connector into place with the two LEDs facing the front of the computer. That's it! I found it to be easier than installing the A501 RAM card into the 500 - about as simple as plugging a cartridge into a Commodore 64.

The module is fitted with one-half, one or two megabytes of low power, zero wait state, autoconfiguring Fast RAM. It owes its small size to the use of one-megabit RAM chips that are soldered directly to the board. According to M.A.S.T., this contributes to overall system reliability (presumably because the chips cannot become loose in their sockets).

Because there are no sockets, it is not recommended that the user attempt to upgrade the MiniMegs Memory Module himself. If you send the unit back to M.A.S.T. with $10 and the extra ZIP-packaged RAM chips, M.A.S.T. will install and burn-in the extra memory. Alternatively, you can purchase the additional memory from the M.A.S.T. factory at their current pricing schedule.

In the event that you're an inveterate hardware hacker who won't let anyone else inside his equipment, M.A.S.T. has thoughtfully included instructions for populating the MiniMegs. As you might expect, this will void your warranty (one year on the board).

The low cost, small size, easy installation, and apparent reliability of the MiniMegs unit make it a good memory expansion solution for those running short of megs.

Schedule Assistant

Voice and Sound Reminders - *Create your own or use NAG PLUS library.*

Perpetual Calendar *-Enter 198 events per day.*

Auto Dialer *-Connects YOU, not your modem.*

AREXX Port *-Commands any timed event or action.*

Notepad *-Click on any word to open text editor.*

Print • Search • And Much More.

Suggested Retail $79.95.

NAG® PLUS 3.0

Gramma software For Your Own Good

Ask your dealer or contact:

Gramma Software
17730 15th Avenue N.E.
Suite 223
Seattle, Washington 98155
Phone #: (206) 363-6417
Fax #: (206) 361-0429

COLOR RIBBONS & PAPER

RIBBONS: Red, Blue, Grn., Brwn., Purple, Yel.

Ribbons Price Each	Black	Color	Heat Transfer
Brother M1109	4.95	5.95	7.00
C. Itoh Prowriter Jr.	7.00	9.00	–
Citizen 120D/180D	5.00	6.00	7.95
Commodore MPS 801	4.50	5.25	5.75
- MPS 802/1526	6.25	7.25	–
- MPS 803	4.95	5.95	7.00
- MPS 1000	3.95	4.95	6.75
- MPS 1200/1250	5.00	6.00	7.95
- 1525	6.50	8.00	–
Epson MX80/LX800	3.75	4.25	6.75
Okidata 82/92	1.75	2.25	4.50
Okidata 182/192	6.50	7.50	6.00
Panasonic K-XP 1080	6.75	7.75	–
Seikosha SP 800/1000	5.25	6.50	7.95
Star SG10	1.75	2.25	4.50
Star NX10/NL10	5.00	6.00	7.95
Star NX1000	4.50	5.50	6.75
Star NX1000C – 4C	–	8.75	–

T-SHIRT RIBBONS (Heat Transfer) – Red, Blue, Grn., Brwn., Purple, Yel., Blk. Call For Price & Availability.

COLOR PAPER

BRIGHT PACK–200 Sheets/50 each color: Red, Blue, Green, Yellow. 9 1/2 x 11 – $11.90/pk.

PASTEL PACK–200 Sheets/50 each color: Pink, Yellow, Blue, Ivory. 9 1/2 x 11 – $11.90/pk.

COLOR BANNER BAND PAPER – 45 ft./roll–$9.95/ea.

For ribbons & paper not listed above, call for price & avail. Price & spec. subject to change w/o notice. Min. order $25.00. Min. S&H $3.50 min. Visa, MC, COD.

RAMCO COMPUTER SUPPLIES
P.O. Box 475, Manteno, IL 60950 U.S.A.
(US) 800-522-6922 • (Canada) 800-621-5444
815-468-8081

News LINK

68030 co-processor board for the Amiga 2000: GVP Inc. announces a 68030 accelerator board for the A2000 capable of running at 16 or 25 MHz. Up to eight Mb of zero wait-state, 32-bit wide, autoconfigured RAM can be added with an optional daughterboard. An optional 68882 FPC (Floating Point Coprocessor) can also be added to the board and clocked from a separate oscillator, allowing the FPC to run at a different speed to the 68030. A built-in hard disk controller provides the board with direct access to the hard disk. The accelerator is available in 16- and 25-MHz configurations. Optional DRAM boards come with 4 Mb or 8 Mb installed. The *A2000-030/16* is available for $849.00 US, the *A2000-030/25* for $999.00 US. The 4-Mb, 32-bit wide RAM board is $1999.00 US.

Removable hard disk technology: GVP Inc. is offering a 44-Mb removable SCSI hard disk drive for the A2000. The half-height drive uses 5.25-in., industry standard cartridges. When matched with one of their SCSI controllers and an autoboot EPROM kit, the A2000 can be booted directly from the removable cartridges. With an average seek time of 25 ms, data transfer rates occur at up to 1.25 Mb/sec. The *Impact SQ44* hard drive lists at $1299.00 US without cartridge. The *SQ400* cartridge is available for $139.00 US. Contact: Grand Valley Products Inc., 255 Plank Rd., Paoliu, PA 19301, USA.

Hardware hacker's delight: The NES *AM-2000A* is an Amiga 2000 prototyping board designed for 300-, 600- and 900-mil chips. Features include: Zorro II and IBM AT card edge connectors and a hole pattern for 'D' pin connectors. According to NES, this prototyping board can be plugged into any Zorro II or IBM slot without fear of contacting or damaging any components on the A2000 motherboard. Cost is $49.95 US, plus shipping and handling. Contact: NES Inc., 4660 S.W. Flower Pl., Portland, OR 97221, USA.

Serial boards for the Amiga: Two eight-port RS-232 serial boards, the *AM201P8M* and the *AM201P8R*, are being offered by NES Inc. The company states that each board has eight channels capable of running at 38.4 Kbps simultaneously, with support for one channel to run at 100 Kbps while other channels are running slower or are disabled. Standard modem control lines are available on each channel. The software is promised to be compatible with Commodore's multiport serial standard for AmigaDOS 1.4. The AM201P8M comes with MIDI rate support and can handle eight MIDI channels concurrently. It supports MIDI transmission rates up to 9.6 kbps accurately. The A2000 will be able to support any three of these two boards at one time, providing up to 25 online serial ports. The AM201P8R is $895.00 US, the AM201P8M (MIDI), $945.00 US. Contact: Nes Inc., 4660 S.W. Flower Pl., Portland, OR 97221, USA.

Amigas go Forth: Version 2.0 of Delta Research's *JForth Professional* has just been released. JForth is a 32-bit, '83 standard Forth. It is an interactive programming environment. Small programs can be compiled in a few seconds allowing for fast development. JForth compiles directly to 68000 machince code for fast execution. Amiga structures can be referenced with automatic sizing and sign extension, and any Amiga library routine can be called by name. Upgrades are available to registered JForth users for $50.00 US. The orginal 'JForth Extras' disk must be included. JForth 2.0 is $179.95 US to first time buyers. Contact: Delta Research, Box 1051, CA 94195, USA.

Modula-2 development system upgrade: ITC has announced the release of version 3.2e of the *M2Amiga Software Development System*. In conjuction with the release, ITC has also produced a source-level debugger/loader. The debugger allows the examination of variable contents at the moment of error. The orginal source code also displays the exact point of any run-time errors. As a loader, the debugger can execute any program immediately after compiling - avoiding the linker for fast development work. Version 3.2e of the M2Amiga system supports the debugger and also includes the new *M2Make* utility. It re-compiles and re-links complete projects by analyzing inter-module dependencies. Supply the name of the top module, and M2Make does the rest. The cost of upgrade is $29.00 US, plus the first page of the current manual. The debugger/loader is available as a separate pakage for $124.00 US. Contact: Interface Technologies Corp., 336 Richmond Ave., Suite 323, Huston, TX 77098, USA.

Inexpensive scanner: Need a scanner? Got a printer? Simply attach SunRize Industries' *IMG Scan* to any dot matrix printer and digitize photographs and drawings. The digitized images can then be saved in IFF format for use with paint, desktop publishing and video programs. IMG Scan works with any dot matrix printer that has adjustable verticle line spacing. Scan at 75, 150, 216, 300, 360 dpi with up to 256 gray levels. IMG Scan is available for $149.00 US. Contact: SunRize Industries, P.O. Box 1453, College Station, TX 77841, USA.

Haitex adds depth to Amiga: With *X-Specs 3D glasses* cabled to a computer, images can be viewed with the depth of real life 3D. The X-Spec glasses control what each eye sees independently by closing and opening an optical shutter at 60 frames a second. The optical shutter employs a high speed liquid crystal. With this technique, the computer can display a flat image to each eye independently and trick the brain into constructing a 3D image. X-Specs have already found their way into the medical field and university chemistry departments. Available for $124.95 US. Contact: Haitex Resources, 208 Carrollton Park, Suite 1207, Carrollton, TX 75006.

Reader Service Card

Fill out this card (please print) or attach your peel-off mailing label, and circle the reader service number for the ads that you'd like more information about (see Advertiser Index below).

Name _____

Company _____

Address _____

_____

Postal/Zip Code _____

1	2	3	4	5	6
7	8	9	10	11	12
13	14	15	16	17	18
19	20	21	22	23	24
25	26	27	28	29	30
31	32	33	34	35	36

TA V2I5

Advertiser	Page	Reader Service #
ASDG Inc.	29	1
Conceptually Advanced Technologies	13	2
Delta Research	17	3
Electronics 2001	29	4
Expressway Software Inc.	12	5
Fishers Computers & Software	29	6
Forethought	68	7
Free Spirit Software Inc.	53	8
Gramma Software	79	9
Inovatronics, Inc.	9	10
Intercomputing Inc.	1	11
King Publishing	68	12
Lake Forest Logic Inc.	12	13
Lattice	Inside Front Cover	14
M.A.S.T.	5	15
Metadigm Inc.	53	16
Newtek Inc.	53	17
Ramco Computers	79	18
The Puzzle Factory	17	19
The Software Shop	25	20
Virtual Reality Laboratories Inc.	53	21

for the AMIGA Transactor

The Magazine for Amiga Programmers

Please print your name and address below (or transfer your mailing label)
Check here for change of address ☐

Name: _____

Address: _____

City: _____ *Province/State:* _____

Zip/Postal: _____ *Phone:* _____ *CompuServe ID:* _____

NOTE: Prepayment required. Purchase orders will be accepted ONLY if accompanied by payment.

Subscribers only: please enter subscription number from mailing label ☐

Subscription Order Section

PLEASE NOTE THAT TRANSACTOR FOR THE AMIGA IS NOW PUBLISHED MONTHLY

	Canada ($Cdn.)			U.S.A. ($US)			Foreign ($US)			Air Mail ($US, overseas only)		
	Mag	Disk*	Both*	Mag	Disk	Both	Mag	Disk	Both	Mag	Disk	Both
12 issues	☐ 35.00	☐ 123.00	☐ 158.00	☐ 27.50	☐ 101.00	☐ 128.50	☐ 38.50	☐ 119.50	☐ 157.50	☐ 73.00	☐ 145.00	☐ 218.00
24 issues	☐ 65.00	☐ 220.00	☐ 270.00	☐ 49.00	☐ 160.00	☐ 190.00	☐ 69.00	☐ 220.00	☐ 290.00	☐ 135.00	☐ 250.00	☐ 390.00

*** Ontario residents please add 8% of disk subscription price for provincial sales tax. No PST on magazine subscriptions.**

Combo Subscription Bonus!

Subscribe to BOTH the magazine and the disk and receive a FREE extra issue on your subscription!

Product Order Section

Ontario residents please add 8% PST on all items

$Cdn.	$U.S.	
☐ 14.95	12.95	Transactor Classic Amiga Disk #1
☐ 14.95	12.95	Transactor Classic Amiga Disk #2

Fish Disks $6.95 US, $7.95 Cdn. each

Note: Disk numbers not yet released may be ordered in advance - simply add numbers to end of list. Disks 57, 80 and 88 are not available.

☐ 1	☐ 2	☐ 3	☐ 4	☐ 5	☐ 6	☐ 7	☐ 8	☐ 9	☐ 10	☐ 11	☐ 12	☐ 13	☐ 14	
☐ 15	☐ 16	☐ 17	☐ 18	☐ 19	☐ 20	☐ 21	☐ 22	☐ 23	☐ 24	☐ 25	☐ 26	☐ 27	☐ 28	
☐ 29	☐ 30	☐ 31	☐ 32	☐ 33	☐ 34	☐ 35	☐ 36	☐ 37	☐ 38	☐ 39	☐ 40	☐ 41	☐ 42	
☐ 43	☐ 44	☐ 45	☐ 46	☐ 47	☐ 48	☐ 49	☐ 50	☐ 51	☐ 52	☐ 53	☐ 54	☐ 55	☐ 56	
☐ 58	☐ 59	☐ 60	☐ 61	☐ 62	☐ 63	☐ 64	☐ 65	☐ 66	☐ 67	☐ 68	☐ 69	☐ 70	☐ 71	
☐ 72	☐ 73	☐ 74	☐ 75	☐ 76	☐ 77	☐ 78	☐ 79	☐ 81	☐ 82	☐ 83	☐ 84	☐ 85	☐ 86	
☐ 87	☐ 89	☐ 90	☐ 91	☐ 92	☐ 93	☐ 94	☐ 95	☐ 96	☐ 97	☐ 98	☐ 99	☐ 100	☐ 101	
☐ 102	☐ 103	☐ 104	☐ 105	☐ 106	☐ 107	☐ 108	☐ 109	☐ 110	☐ 111	☐ 112	☐ 113	☐ 114	☐ 115	
☐ 116	☐ 117	☐ 118	☐ 119	☐ 120	☐ 121	☐ 122	☐ 123	☐ 124	☐ 125	☐ 126	☐ 127	☐ 128	☐ 129	
☐ 130	☐ 131	☐ 133	☐ 134	☐ 135	☐ 136	☐ 137	☐ 138	☐ 139	☐ 140	☐ 141	☐ 142	☐ 143	☐ 144	
☐ 145	☐ 146	☐ 147	☐ 148	☐ 149	☐ 150	☐ 151	☐ 152	☐ 153	☐ 154	☐ 155	☐ 156	☐ 157	☐ 158	
☐ 159	☐ 160	☐ 161	☐ 162	☐ 163	☐ 164	☐ 165	☐ 166	☐ 167	☐ 168	☐ 169	☐ 170	☐ 171	☐ 172	

The Transactor for the *AMIGA* Disk $12.95 US, $14.95 Cdn. each

(these disks correspond to issues of **Transactor for the *AMIGA***)

☐ Vol. 1, Iss. 1 ☐ Vol. 1, Iss. 2 ☐ Vol. 1, Iss. 3 ☐ Vol. 1, Iss. 4 ☐ Vol. 1, Iss. 5 ☐ Vol. 1, Iss. 6
☐ Vol. 2, Iss. 1 ☐ Vol. 2, Iss. 2 ☐ Vol. 2, Iss. 3 ☐ Vol. 2, Iss. 4 ☐ Vol. 2, Iss. 5

Add flat $2.00 shipping & handling for any number of items ordered from above box.

Back Issues: $4.50 U.S., $4.50 Cdn. (filled box=SOLD OUT)

☐ Vol. 1, Iss. 1 ■ Vol. 1, Iss. 2 ☐ Vol. 1, Iss. 3 ☐ Vol. 1, Iss. 4 ☐ Vol. 1, Iss. 5 ☐ Vol. 1, Iss. 6
☐ Vol. 2, Iss. 1 ☐ Vol. 2, Iss. 2 ☐ Vol. 2, Iss. 3 ☐ Vol. 2, Iss. 4

Ontario residents please add 8% provincial sales tax for items ordered from this box.

☐ Visa ☐ MasterCard ☐ Cheque/MO enclosed Cheque #_____ Dated ___/___/___ Amount _____

Acct. # _____ Expires ___/___ Signature _____

Please allow 6-8 weeks for delivery in the U.S., 4-6 weeks in Canada.

FOLD ALONG SCORES WITH CORRECT REPLY CARD ON OUTSIDE. USE TAPE, PLEASE - NO STAPLES OR GLUE.

NO POSTAGE NECESSARY IF MAILED IN THE UNITED STATES

BUSINESS REPLY MAIL
FIRST CLASS PERMIT NO. 390 BUFFALO, NY
POSTAGE WILL BE PAID BY ADDRESSEE

Transactor
P.O. Box 338 Station 'C'
Buffalo, NY, 14209-9990

place stamp here

Transactor
10-85 West Wilmot Street
Richmond Hill, Ontario, Canada
L4B 1K7

Business Reply Mail
No Postage Stamp Necessary
If mailed in Canada

Postage will be paid by:

Canada Post Postes Canada 46487

Transactor
501 Alden Road
P.O. Box 3250
Markham Industrial Park
Markham, Ontario
L3R 9Z9

Advertising in Transactor for the Amiga

Interested in advertising in *TransAmi?* Call Mike Grantham at 416/764-5274

THE WORLD OF COMMODORE

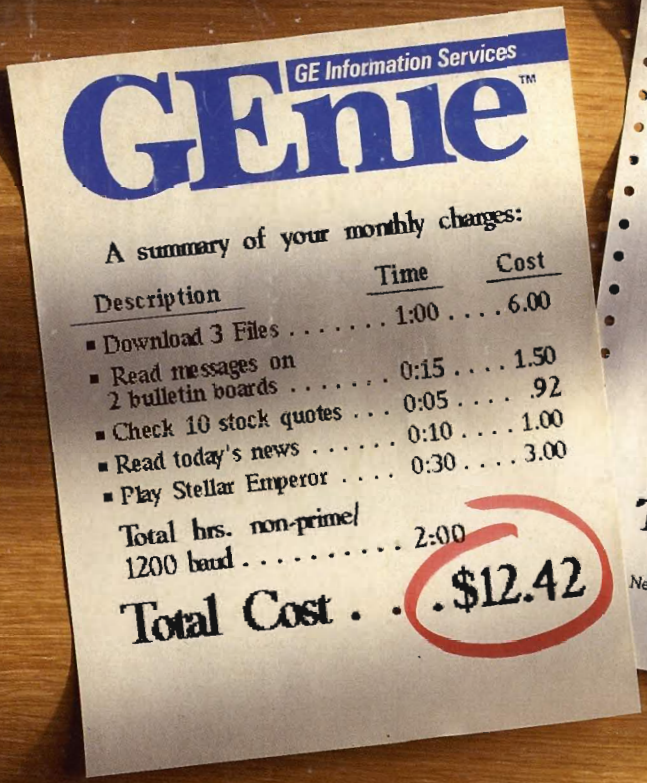
Welcome to a spectacular world of Commodore computing — a world devoted to the **Amiga, C-64, C-128 and Commodore PCs.** You'll discover the software you've always wanted to try, plus amazing, new programs. You'll find printers and plotters. Modems and monitors. Disk drives and joy sticks. Lasers and light pens. MIDI and mice. All the big and little stuff that make computing more productive, more creative — more fun!
And some of the best bargains you'll find anywhere!

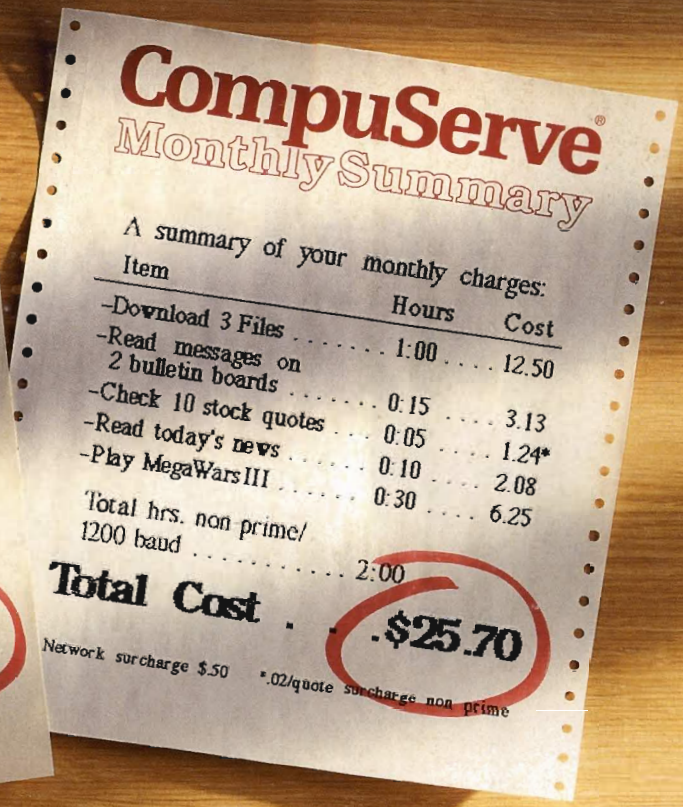
It's all in one place — at the 2nd annual North-Eastern World of Commodore. Whether you compute for business or fun, at home or school, you can't miss this computer show!

THE WORLD OF COMMODORE
September 22, 23 and 24, 1989

Fri. noon–8pm/Sat.&Sun. 10am-5pm
Admission $10 Students/Seniors $8
Includes seminars & stage demonstrations

Exhibitors contact: The *Hunter Group* **(416) 595-5906** Fax (416) 595-5093. Produced in association with Commodore Business Machines.





It doesn't take a computer to figure out the difference.

The difference between the GEnie™ service and CompuServe® could make a big difference to you. Here's why. GEnie's rate for 1200 baud access is just $6 per non-prime hour.* Theirs is more than twice as much. Which means that with GEnie you can stay online longer for a whole lot less.

And that means more time to enjoy more of GEnie's services. Like valuable computer RoundTables. Exciting multi-player games. Useful financial information and much more. All at hourly rates designed to allow you to spend more time online. Not more money.

And with GEnie signing up is as easy as one, two, three.

(1) Set your modem for local echo (half duplex), 300 or 1200 baud.
(2) Dial 1-800-638-8369. When connected, you just enter HHH.
(3) At the U#= prompt simply enter XTX99645,GEnie then RETURN.

And have a major credit card or your checking account number ready. For information in the U.S. or Canada, call 1-800-638-9636. Or write GE Information Services, 401 North Washington, Rockville, MD 20850.

We bring good things to life.

**Applies only in U.S. Mon.-Fri. 6PM-8AM local time and all day Sat., Sun., and national holidays. Subject to service availability. Some services may be subject to a surcharge. Prices effective May 1, 1989.*